Tuning af CPH STLs komponentstrukturer for smeltbare prioritetskøer.

-

Kildetekst Appendix

Asger Bruun

September 2010. Rettelse september 2010*.

Contents

*Tilføjet glemte kildetekster (Code/CPHSTL_modified). Slettet uddaterede kildetekster. Øget tekstbredde med 25% for at reducere word wrapping. Slettet udkommenterede blokke og uddateret fejlsporing.

A Code

A.1 binary_number_system.hpp

```
#ifndef _CPHSTL_BINARY_NUMBER_SYSTEM_H_
#define _CPHSTL_BINARY_NUMBER_SYSTEM_H_

/*
Desc: Vuillemin style binary number system.
Auth: Asger Bruun 2009-2010.
Ref:  Jean Vuillemin. A data structure for manipulating priority queues.
      Communications of the ACM 21 (1978), 309-315.
*/

namespace cphstl {

  template<typename C, typename E>
  // where E and C have the same type of comparator and value.
  class binary_number : public E::root_list_type {
    binary_number();
    binary_number(binary_number const&);
    binary_number& operator = (const binary_number&);

  public:

    typedef typename E::comparator_type comparator_type;
    typedef typename E::root_list_type root_list_type;
    typedef typename root_list_type::back_inserter back_inserter;
    typedef typename E::size_type size_type;
    typedef typename E::pointer pointer;

    //would like protected:

    class bit_scanner {
      size_type current_, bits_left_;

    public:

      bit_scanner(size_type const& d) : bits_left_(d) {
        if(bits_left_ == 0) current_ = 0;
        else {
          current_ = 1;
          next_bit();
        }
      }

      size_type current() { return current_; }

      void next_bit() {
        assert(current_ != 0);
        while(true) {
          if(((*this).bits_left_ & 1) != 0) { --(*this).bits_left_; break; }
          else if((*this).bits_left_ == 0) { current_ = 0; break; }
          else { (*this).bits_left_ >>= 1; current_ <<= 1; }
        }
      }

    private:

      friend std::ostream & operator <<
        (std::ostream & os, bit_scanner const& bs)
      {
        bit_scanner tmp(bs);
        while(tmp.current()) {
          os << tmp.current() << " "; tmp.next_bit();
        }
        return os;
      }
    };

  public:

    enum { number_system = 2 };
```

```
70
71      typedef binary_number<C,E> fast_number_type;
72      typedef typename fast_number_type::bit_scanner fast_bit_scanner;
73
74      explicit binary_number(pointer list, size_type size)
75        : root_list_type(list), accu(size) {
76          assert(is_valid());
77      }
78
79      bool is_valid() /*const*/ { return root_list_type::is_valid(accu); }
80
81      size_type size() const { return accu; }
82      size_type max_size() const {
83        // Note: It is clear to the LWG that the value returned by
84        //       max_size() can't change from call to call.
85        // Ref:  www.open-std.org/jtc1/sc22/wg21/docs/lwg-closed.html#197
86        return std::numeric_limits<size_type>::max() / E::footprint();
87      }
88
89      void swap(binary_number<C,E> & h2) {
90        assert((*this).is_valid()); assert(h2.is_valid());
91        root_list_type::swap(h2);
92        std::swap(accu, h2.accu);
93      }
94
95      //following: temporarely made public because of problem with friend.
96      size_type accu;
97
98    protected:
99
100     static comparator_type comparator;
101
102   private:
103
104     // adder
105     typedef typename root_list_type::adder_access_type adder_access_type;
106     typedef typename root_list_type::adder_access_ref adder_access_ref;
107     static void accept_carry(back_inserter & nxt, pointer& c) {
108       nxt.add(c); c = 0;
109     }
110     static void accept_res(back_inserter & nxt, adder_access_ref l) {
111       nxt.add(l.skip());
112     }
113     static void prop_carry(pointer& c, adder_access_ref l) {
114       c = (*l.eject()).join(c, comparator);
115     }
116     static void prop_carry2(pointer& c, adder_access_ref l) {
117       c = (*c).join(l.eject(), comparator);
118     }
119     static void construct_carry(pointer& c, adder_access_ref l1, adder_access_ref l2) {
120       c = (*l1.eject()).join(l2.eject(), comparator);
121     }
122
123   protected:
124
125     size_type remove_root(pointer r) {
126       // Desc: remove root from root list and return its size.
127       // Note: bidirectional root list can't help skip scanning
128       // :     because size calculaiton is needed here.
129       assert((*r).is_root());
130       pointer y((*this).begin()), z((*this).end());
131       bit_scanner bs((*this).accu);
132       while(y != r) {
133         bs.next_bit(); z = y; y = next_get(y);
134       }
135       size_type delta(bs.current());
136       root_list_type::remove_after(z);
137       (*this).accu -= delta;
138       return delta;
139     }
140
141     void add(binary_number& b) {
142       // T(n,m) = O(min(lg m, lg n)) amortized.
143
```

```
144         assert((*this).is_valid()); assert(b.is_valid());
145
146         if(b.accu > (*this).accu) (*this).swap(b); // pick root list with highest digit.
147
148         adder_access_type r1(*this), r2(b); // ref, not copy.
149         size_type n1((*this).accu), n2(b.accu);
150         pointer c(0);
151         root_list_type res(0);
152         back_inserter nxt(res);
153         while((n1 != 0 && n2 != 0) || (c != 0)) {
154           switch((c == 0?0:4) + ((n2 & 1) << 1) + (n1 & 1)) {
155             case 0 /*000*/: /*output digit zero*/; break;
156             case 1 /*001*/: accept_res(nxt, r1); break;
157             case 2 /*010*/: accept_res(nxt, r2); break;
158             case 3 /*011*/: construct_carry(c, r1, r2); break;
159             case 4 /*100*/: accept_carry(nxt, c); break;
160             case 5 /*101*/: prop_carry(c, r1); break;
161             case 6 /*110*/: prop_carry2(c, r2); break;
162             case 7 /*111*/: accept_res(nxt, r1); prop_carry(c, r2); break;
163             default: assert(0); break;
164           }
165           n1 >>= 1; n2 >>= 1;
166         }
167         if(n1 != 0) {
168           assert(n2 == 0); assert(r2.empty());
169           accept_res(nxt, r1);
170           (*this).clear();
171         } else if(n2 != 0) {
172           assert(r1.empty());
173           accept_res(nxt, r2);
174           b.clear();
175         } else {
176           assert(r1.empty());
177           assert(r2.empty());
178         }
179         (*this).accu += b.accu;
180         b.accu = 0;
181         (*this).root_list_type::swap(res);
182
183         assert((*this).is_valid()); assert(b.is_valid());
184       }
185
186       void increment(pointer p) {
187         // Add one elemenet and increment size.
188         // Only every second contiguous insert does meld.
189         assert(is_valid());
190         assert(p != 0); assert((*p).height() == 0);
191         assert((*p).is_root());
192
193         if(((*this).accu & 1) == 0) {
194           inject(p);
195           ++accu;
196         } else {
197           binary_number hs(p, 1);
198           add(hs);
199         }
200         assert(is_valid());
201       }
202
203       pointer decrement() {
204         // Remove one elemenet and decrement size.
205         // Meld is never needed because the tree extracted
206         // is always the least significant digit.
207         assert((*this).is_valid()); assert(!(*this).empty());
208
209         pointer p(root_list_type::eject());
210         pointer t((*p).most_significant_child());
211         if(t != 0) { // !(*p).is_leaf()
212           size_type n;
213           pointer r((*p).construct(n, constant<bool,false>()));
214           inject_range(r,t);
215         }
216         --accu;
217         assert(is_valid());
```

```
218         assert((*p).is_root());
219         assert((*p).height() == 0);
220         return p;
221       }
222
223       void promoted_to_root(pointer replacement, pointer d) {
224         root_list_type::promoted_to_root(replacement,d);
225         assert((*this).is_valid());
226       }
227
228
229
230     private:
231
232       friend std::ostream & operator <<
233         (std::ostream & os, binary_number const& hs)
234       {
235         size_type n(hs.accu);
236         os << "size(" << n << ")␣";
237         pointer r(const_cast<binary_number &>(hs).begin());
238         size_type h(0);
239         while(n != 0) {
240           if((n & 1) != 0) {
241             os << "[" << h << ":" << (*r) << "]␣";
242             if(r != 0) r = (*r).next_root_get();
243           }
244           n >>= 1; ++h;
245         }
246         os << "\n";
247         return os;
248       }
249     };
250     template<typename C, typename E>
251     typename binary_number<C,E>::comparator_type binary_number<C,E>::comparator = C();
252   } // namespace cphstl
253   #endif
```

A.2 binomial_node.hpp

```
1   /*
2   Desc: A traditional binomial three-pointer node.
3         Probably naive because of its big penalty in promote.
4         This is the node type V of Brown.
5
6   Auth: Jyrki Katajainen, Asger Bruun © 2009
7
8   Ref : M.R. Brown, Implementation and analysis of binomial queue
9         algorithms, SIAM Journal on Computing 7 (1978), 298-319.
10
11
12  *   o------
13  *   : : :  \
14  *   o-o-o---o--
15  *     | :\   : :\
16  *     o o-o  o-o-o
17  *         |    | :\
18  *         o    o o-o
19  *                |
20  *                o
21  */
22
23  #ifndef _CPHSTL_PQFW_BINOMIAL_NODE_H_
24  #define _CPHSTL_PQFW_BINOMIAL_NODE_H_
25
26  #include "assert.h++" // assert
27  #include <cstddef> // std::size_t
28  #include <string> // std::string
29  #include <sstream> //?
30  #include <iostream> // std::ostream + std::ostringstream
31  //#include <tuple>
32
33  #include "root_list.hpp"
34
```

```
35  namespace cphstl {
36    namespace pqfw_node {
37      class binomial_node_base {
38        typedef binomial_node_base N;
39
40        union {
41          void* owner_;
42          N* sibling_; // ~ WHN-left
43        };
44        N* parent_; // ~ WHN-distinguished-ancestor
45        N* child_; // ~ WHN-right
46
47        binomial_node_base(N const&);
48        N& operator = (N const&);
49
50      public:
51
52        typedef binomial_node_base node_base;
53        typedef std::size_t size_type;
54        typedef unsigned char height_type; // assert(size_of(void*) <= (256/8));
55        // typedef root_list<N> root_list_type;
56
57        static N* const nil;
58
59        struct hole_type {
60         // Desc: Defines the perimeter of an internal sub heap.
61          N* da;
62          N* parent;
63          union {
64            N* sibling;
65            void* owner;
66          };
67          N** ingoing;
68
69          hole_type() : da(0), parent(0), sibling(0), ingoing(0) {}
70
71          hole_type(N* n, bool auto_splice_out, bool forced = false)
72            : da((*n).distinguished_ancestor())
73            , parent((*n).parent_), sibling((*n).sibling_)
74          {
75            assert(n != 0);
76            N* p((*n).parent_);
77            if(p == 0) ingoing = 0;
78            else {
79              N* ps((*n).prev_sibling(p));
80              ingoing = ps == 0?&((*p).child_):&((*ps).sibling_);
81            }
82            if((auto_splice_out && do_splice()) || forced) {
83              assert((*n).is_valid());
84              if(careful) {
85                (*n).parent_ = 0;
86                (*n).sibling_ = 0;
87              }
88            }
89          }
90          bool is_root() const { return ingoing == 0; }
91          bool do_splice() { return !is_root(); }
92          void splice_in(N* n, bool forced = false) {
93            if(!(do_splice() || forced)) return;
94            (*n).parent_ = parent;
95            (*n).sibling_ = sibling;
96            if(ingoing != 0) (*ingoing) = n; //???
97          }
98        };
99        hole_type splice_out(bool forced = false) {
100         hole_type hole(this,true,forced);
101         return hole;
102       }
103       template <typename H>
104       void splice_in(hole_type& hole, H&, bool forced = false) {
105         hole.splice_in(this, forced);
106       }
107
108       // structors
```

```
109
110         binomial_node_base() : sibling_(0), parent_(0), child_(0) {}
111
112         // properties.
113
114         static size_type footprint() { return sizeof(N); }
115
116         static bool is_spooky(N const* p) { return 0<(long long)p && (long long)p <1000; }
117         bool is_valid() const {
118           bool node_is_valid = this != nil
119             && !(is_spooky(this) || is_spooky(parent_)
120              || is_spooky(child_) || is_spooky(sibling_));
121           assert(node_is_valid);
122           return(node_is_valid);
123         }
124
125         bool is_other_root() const { assert(is_valid()); return parent_ == nil; }
126         bool is_root() const { assert(is_valid()); return parent_ == nil; }
127         bool is_leaf() const { assert(is_valid()); return child_ == nil; }
128         bool is_marked() const { assert(is_valid()); return false; } // ??
129
130         void* owner() const { assert((*this).is_root()); return owner_; }
131         void*& owner() { assert((*this).is_root()); return owner_; }
132
133         N* root() const {
134           // Desc: find the root. T(n): O(lg(n)). ES: no throw.
135           assert(is_valid());
136           N* p((N*) this);
137           while (! (*p).is_root()) { p = (*p).parent_; }
138           return p;
139         }
140
141         N* distinguished_ancestor() const {
142           assert(is_valid()); return parent_;
143         }
144         N* distinguished_descendant() const {
145           assert(is_valid());
146           return (*this).is_root()?child():sibling();
147         }
148
149         N* most_significant_child() {
150           // Desc: the potentiel return of a split before doing it,
151           //       (which is also the tail of a potential construct).
152           assert(is_valid());
153           return (*this).child_;
154         }
155
156
157         N* successor() const {
158           // Desc: "child first"-ordered traversal.
159           // T(n): O(1). ES: no throw.
160           assert(is_valid());
161           if((*this).child_ != nil) return child_;
162           else if (parent_ != nil && !(*parent_).is_root())
163             return (*parent_).sibling();
164           else return nil;
165         }
166
167         height_type height() const {
168           // Desc: Slow height, usefull for assertions. ES: no throw.
169           assert(is_valid());
170           height_type h(0);
171           for(N* c(child_); c != 0; c = (*c).child_) { ++h; }
172           return h;
173         }
174
175         bool tree_valid() const {
176           if(this == 0) return false;
177
178           if(child() != 0) {
179             N* p((*child()).parent());
180             if(p != this)
181               return false;
182             if(!(*child()).tree_valid())
```

```
183             return false;
184           if(sibling() != 0) {
185             if(!(*sibling()).tree_valid())
186               return false;
187           }
188           return true;
189         } else return true;
190       }
191
192       // operations.
193
194       N* join(N* weaker) {
195         // Desc: join with a weaker tree. T(n): O(1). ES: no throw.
196         // Note: the owner should ensure "assert((*this).height() == (*weaker).height())";
197         assert(!careful || is_root());
198         right_join(weaker);
199         return this;
200       }
201
202       N* split() {
203         // Desc: split weaker tree. T(n): O(1). ES: no throw.
204         assert(!careful || is_root());
205         return right_split();
206       }
207
208       template <class Boolean>
209       N* construct(size_type& n, const Boolean calc_size) {
210         // Desc: Vuillemin "construct". T(n): O(lg n). ES: no throw.
211         //       Isolate the root from its childs.
212         // Ret:  A root list in the reverse order of an recursive split
213         //       + total number of nodes including the root.
214         assert(!careful || is_root());
215         assert(is_valid());
216         N* r(0);
217         if(calc_size) n = 1;
218         for(N* c(child_); c != 0; ) {
219           N* d((*c).sibling_);
220           (*c).parent_ = 0;
221           (*c).sibling_ = r;
222           r = c;
223           if(calc_size) n <<= 1;
224           c = d;
225         }
226         child_ = 0;
227         return r;
228       }
229
230       N* release_root() {
231         // Desc: Vuillemin construct with reverse root list ordering.
232         N* q((*this).child_);
233         (*this).child_ = 0;
234         (*q).parent_ = 0;
235         return q;
236       }
237
238       N* release_subheap() {
239         // Desc: disconnect reverse root list tail.
240         N* q((*this).sibling_);
241         (*this).sibling_ = 0;
242         if (q != 0) (*q).parent_ = 0;
243         return q;
244       }
245
246       N* swap(N* p) {
247         // general swap not needed or supported.
248         assert(p == (*this).distinguished_ancestor());
249         return promote(p);
250       }
251
252       void swap_roots(N* q) {
253         assert(is_root()); assert((*q).is_root());
254         N* p(this);
255         std::swap((*p).child_, (*q).child_);
256         for(N* c = (*q).child_; c != nil ; c = (*c).sibling_) (*c).parent_ = q;
```

```
257           for(N* c = (*p).child_; c != nil ; c = (*c).sibling_) (*c).parent_ = p;
258         }
259
260         N* promote_alt(N* p) {
261           // Vuillemin compatible.
262           promote(p);
263           return (*this).distinguished_ancestor();
264         }
265
266         N* promote(N* p) {
267           // CPH STL compatible.
268           /*
269           * Desc: exchange position with distinguished_ancestor.
270           * T(n): O(lg(n)). ES: no throw.
271           * Note: binomial parent = distinguished_ancestor.
272           *
273           * 4 cases (out of 6), for node N:
274           *   o------          G------          G------          G------
275           *   : : :  \         : : :  \         : : :  \         : : :  \
276           *   o-o-o---G--      o-o-o---P--      o-o-P---F--      o-o-P---F--
277           *     : :\   : :\      : :\   : :\      : :\   : :\      : :\   : :\
278           *    o  o-o  o-o-P    o  o-o  N-o-S    o  o-N  o-o-o    o  N-S  o-o-o
279           *         |    : :\        |    : :\        |    : :\        |    : :\
280           *        o     o  o-N      o    o  o-o      o    o  o-o      o    o  o-o
281           *                  |             |              |              |
282           *                 o             o              o              o
283           *    P == F && N == S     P == F && N != S    P != F && N == S    P != F && N != S
284           *    (parent P, oldest sibling S, grandparent G
285           *     , grand first child F)
286           */
287           assert(is_valid()); assert(parent_ != nil);
288           assert(p == (*this).distinguished_ancestor());
289           N* c((*p).child_);
290
291           std::swap((*this).sibling_, (*p).sibling_);
292           (*this).parent_ = (*p).parent_;
293           (*p).child_ = (*this).child_;
294
295           if(c == this) { // update new children of this (N == S).
296             (*this).child_ = p;
297             for(c = p; c != nil ; c = (*c).sibling_) (*c).parent_ = this;
298           } else { // update new children of this and locate youngest older (N != S).
299             (*this).child_ = c;
300             while((*c).sibling_ != this) { (*c).parent_ = this; c = (*c).sibling_; }
301             (*c).sibling_ = p;
302             while(c != nil) { (*c).parent_ = this; c = (*c).sibling_; }
303           }
304           (*p).parent_ = this;
305
306           // update new children of p
307           for(c = (*p).child_; c != nil ; c = (*c).sibling_) (*c).parent_ = p;
308
309           if((*this).parent_ != nil) { // grandparent exists.
310             c = (*(*this).parent_).child_;
311             if(c == p) (*(*this).parent_).child_ = this; // (P == F).
312             else { // locate youngest older sibling of parent (P != F).
313               while((*c).sibling_ != p) c = (*c).sibling_;
314               (*c).sibling_ = this;
315             }
316           }
317           return this;
318         }
319
320         std::string str() const { return (*this).str<N>(); }
321
322       protected:
323
324         template <typename D>
325         std::string str() const {
326           std::stringstream os;
327           if(this == 0) os << "nil";
328           else {
329             N* x((*this).child());
330             if(x != 0) {
```

```
331             assert((*x).parent() == this);
332             os << "(" << (*static_cast<D *const>(x)).str();
333             for(x = (*x).sibling(); x != 0; x = (*x).sibling())
334               os << "," << (*static_cast<D *const>(x)).str();
335             os << ")";
336           }
337         }
338         return os.str();
339       }
340
341       N* sibling() const {
342         // Note: decide if using sibling on a root is legal or not.
343         assert(is_valid());
344         return ((*this).parent_ != 0)?sibling_:0;
345       }
346       N* child() const { assert(is_valid()); return child_; }
347       N* parent() const { assert(is_valid()); return parent_; }
348
349     private:
350
351       // subtree split and join.
352
353       N* right_join(N* weaker) {
354         // Desc: join with a weaker tree. T(n): O(1). ES: no throw.
355         assert(is_valid());
356         assert(!careful || (*weaker).is_root());
357         assert(this != weaker);
358         (*weaker).parent_ = this;
359         (*weaker).sibling_ = (*this).child_;
360         (*this).child_ = weaker;
361         return this;
362       }
363
364       N* right_split() {
365         // Desc: split weaker tree. T(n): O(1). ES: no throw.
366         assert(is_valid()); assert(child_ != nil);
367         N* b((*this).child_);
368         (*this).child_ = (*b).sibling_;
369         (*b).sibling_ = nil;
370         (*b).parent_ = nil;
371         return b;
372       }
373
374       // splice subtree helpers.
375
376       N* wrong_join(N* younger) {
377         // Desc: insert sibling. T(n): O(1). ES: no throw.
378         assert(is_valid()); assert((*younger).is_root()); assert(this != younger);
379         (*younger).parent_ = (*this).parent_;
380         (*younger).sibling_ = (*this).sibling_;
381         (*this).sibling_ = younger;
382         return this;
383       }
384
385       N* wrong_split() {
386         // Desc: remove next sibling. T(n): O(1). ES: no throw.
387         assert(is_valid()); assert(sibling_ != nil);
388         N* b((*this).sibling_);
389         (*this).sibling_ = (*b).sibling_; (*b).sibling_ = nil; (*b).parent_ = nil;
390         return b;
391       }
392
393       N* prev_sibling(N* known_parent) const { // "light weak and binomial identical code"
394         // Desc: locate older sibling from parent or return 0.
395         //       (use prev_sibling(parent())
396         assert(is_valid()); assert(known_parent == (*this).parent());
397         if (known_parent == 0) return 0;
398         else {
399           N* c((*known_parent).child_);
400           if(c == this) c = 0;
401           else while((*c).sibling() != this) c = (*c).sibling();
402           return c;
403         }
404       }
```

```
405       };
406       binomial_node_base * const binomial_node_base::nil(0);
407
408       template <>
409       struct node_traits<binomial_node_base> { enum { has_splice = 1 }; enum { has_owner = 1
              }; enum { is_root_listable = 1 }; };
410     }
411   }
412   #endif
```

A.3 bit_manipulation.h++

```
1   ï■¿/*
2   Desc: Machine dependant bit manipulation (in terms of Intel).
3   Auth: Asger Bruun 2009-2010.
4   Hist: Imported from priority queue Vuilllemin.
5   Warn: This is fragile cross platform code.
6   Impl: Simple overloading might leave our compiler in conflict when promoting a given type
        ,
7         eg. if a question about signed/unsigned is more important that one about short/long
              .
8         This implementation is directed by distinct word widths in number of bits with
9         direct hardware support, to allow for operating efficiently on widths lower
10        than of std:size_t too. The optimal methods for a given bit width are selected
11        automatically using class specialisation (a complicated work around on the problem
12        that C++ does not allow for partial function template specialization).
13  Note: The specialisations are directed against sizes in powers of 2 only,
14        that is no promotion is attempted for other sizes.
15  */
16
17  #ifndef _CPHSTL_BIT_MANIPULATION_HPP_
18  #define _CPHSTL_BIT_MANIPULATION_HPP_
19
20  #include "assert.h++"
21  #include <climits> // CHAR_BIT
22  #include <limits> // CHAR_BIT
23
24  #ifdef _MSC_VER
25  // Platform: Microsoft C, Windows.
26  // Ref: msdn.microsoft.com/en-us/library/26td21ds(v = VS.90).aspx
27  #include <intrin.h>
28  #include "bit_manipulation_cpu_config.hpp"
29  #elif defined(__GNUC__)
30  // Platform: GCC - no extra configuration includes needed.
31  // Ref: gcc.gnu.org/onlinedocs/gcc-4.4.0/gcc/Other-Builtins.html#index-
        g_t_005f_005fbuiltin_005fffs-2894
32  #endif // GNUC
33
34  namespace cphstl {
35    // Quote: www.open-std.org/JTC1/SC22/wg14/www/docs/n1425.pdf
36    //   7.18 Common definitions <stddef.h>
37    //   ... The types are
38    //   ptrdiff_t which is the signed integer type of the result of subtracting two
          pointers;
39    //   size_t which is the unsigned integer type of the result of the sizeof operator...
40    typedef std::size_t word_type;
41    typedef std::ptrdiff_t diff_type;
42
43
44    template <typename T, int bit_width>
45    struct bit_man; // specialised implementation forward
46
47
48    // -----  User interface.
49
50    template <typename T>
51    T trailing_zeros(const T n) {
52      // Desc: number of least significant zero bits
53      //       (position of least significant nonzero bit).
54      assert(n != 0); assert(std::numeric_limits<T>::is_integer);
55      return bit_man<T, sizeof(T)*CHAR_BIT>::trailing_zeros(n);
56    }
57    template <typename T>
```

```
58    T bit_scan_reverse(const T n) {
59      // Desc: position of most significant nonzero bit.
60      assert(n != 0); assert(std::numeric_limits<T>::is_integer);
61      return bit_man<T,sizeof(T)*CHAR_BIT>::bit_scan_reverse(n);
62    }
63    template <typename T>
64    T leading_zeros(const T n) {
65      // Desc: position of least significant nonzero bit.
66      // Warn: the result of using leading zeros on a wrong data type,
67      //       eg. on a long where it was actually a char, is catastrophic.
68      assert(n != 0); assert(std::numeric_limits<T>::is_integer);
69      return bit_man<T,sizeof(T)*CHAR_BIT>::leading_zeros(n);
70    }
71    template <typename T>
72    T pop_cnt(const T n) {
73      // Desc: number of nonzero bits.
74      assert(std::numeric_limits<T>::is_integer);
75      return bit_man<T,sizeof(T)*CHAR_BIT>::pop_cnt(n);
76    }
77    template <typename T, typename S>
78    bool bit_test(T& n, const S pos) {
79      // Desc: test bit at a given position.
80      assert(std::numeric_limits<T>::is_integer);
81      assert(std::numeric_limits<S>::is_integer);
82      return bit_man<T,sizeof(T)*CHAR_BIT>::bit_test(n, pos);
83    }
84
85    // Jyrki names:
86    template<typename T> T population_count(T n) {
87      return pop_cnt(n);
88    }
89
90
91    // -----  Generic platform inefficient code.
92
93    template <typename T, int bit_width>
94    struct bit_man_generic {
95      static T trailing_zeros(T n){
96        assert(n != 0);
97        T b(0);
98        while (n % 2 == 0) { n >>= 1; ++b; }
99        return b;
100     }
101     static T bit_scan_reverse(T n) {
102       assert(n != 0);
103       T b(0);
104       n >>= 1;
105       while (n != 0) { n >>= 1; ++b; }
106       return b;
107     }
108     static T leading_zeros(T n) {
109       assert(n != 0);
110       return (sizeof(T)*CHAR_BIT) - bit_man<T,bit_width>::bit_scan_reverse(n) - 1;
111     }
112     static T pop_cnt(T n) {
113       T b(0);
114       while (n != 0) { b += (n % 2); n >>= 1; }
115       return b;
116     }
117     template<typename S>
118     static bool bit_test(T& n, const S pos) {
119       assert(pos >= 0);
120       assert(pos < (signed)(sizeof(T)*CHAR_BIT));
121       return (n & (T(1) << pos)) != 0;
122     }
123   };
124
125   template <typename T, int bit_width>
126   struct bit_man : public bit_man_generic<T,bit_width> {};
127
128
129   // -----  Platform dependant efficient code.
130
131 #if defined(_MSC_VER) && (defined(_M_IX86) || defined(_M_X64)) // MSC 32/64-bit Intel/AMD
```

```
132 #pragma intrinsic(_bittest,_BitScanForward,_BitScanReverse,__popcnt)
133 #if defined(_M_X64) // 64-bit
134 #pragma intrinsic(_bittest64,_BitScanForward64,_BitScanReverse64,__popcnt64)
135
136   template <typename T>
137   struct bit_man<T,64> : public bit_man_generic<T,64> {
138     static unsigned long trailing_zeros(T /*unsigned long long*/ n) {
139       assert(n != 0);
140       unsigned long index;
141       /*unsigned char is_non_zero  = */ _BitScanForward64(&index, n);
142       return index;
143     }
144     static unsigned long bit_scan_reverse(T /*unsigned long long*/ n) {
145       assert(n != 0);
146       unsigned long index;
147       /*unsigned char is_non_zero  = */ _BitScanReverse64(&index, n);
148       return index;
149     }
150 #ifdef CPU_HAS_LZ_CNT
151     static unsigned long long leading_zeros(T /*unsigned long long*/ n) {
152       assert(n != 0); assert(cpu_has_lz_cnt());
153       return __lzcnt64(n);
154     }
155 #endif
156 #ifdef CPU_HAS_POP_CNT
157     static unsigned long long pop_cnt(T /*unsigned long long*/ n) {
158       assert(cpu_has_pop_cnt());
159       return __popcnt64(n);
160     }
161 #endif
162     template<typename S>
163     static bool bit_test(T /*unsigned long long*/ & n, S /*unsigned long long*/ pos) {
164       return _bittest64((__int64*) &n, pos) != 0;
165     }
166   };
167
168 #endif
169
170   // 32-bit
171
172   template <typename T>
173   struct bit_man<T,32> : public bit_man_generic<T,32> {
174     static unsigned long trailing_zeros(const T /*unsigned long*/ n) {
175       assert(n != 0);
176       unsigned long index;
177       /*unsigned char is_non_zero  = */ _BitScanForward(&index, n);
178       return index;
179     }
180     static unsigned long bit_scan_reverse(const T /*unsigned long*/ n) {
181       assert(n != 0);
182       unsigned long index;
183       /*unsigned char is_non_zero  = */ _BitScanReverse(&index, n);
184       return index;
185     }
186 #if defined(CPU_HAS_LZ_CNT)
187     static unsigned long leading_zeros(T /*unsigned long*/ n) {
188       assert(n != 0); assert(cpu_has_lz_cnt());
189       return __lzcnt(n);
190     }
191 #endif
192 #ifdef CPU_HAS_POP_CNT
193     static unsigned int pop_cnt(T /*unsigned long*/ n) {
194       assert(cpu_has_pop_cnt());
195       return __popcnt(n);
196     }
197 #endif
198     template<typename S>
199     static bool bit_test(T& n, const S /*unsigned long*/ pos) {
200       return _bittest((long*) &n, pos) != 0;
201     }
202   };
203
204   // 16-bit
205
```

```
206     template <typename T>
207     struct bit_man<T,16> : public bit_man_generic<T,16> {
208   #ifdef CPU_HAS_LZ_CNT
209       static unsigned short leading_zeros(T /*unsigned short*/ n) {
210         assert(n != 0); assert(cpu_has_lz_cnt());
211         return __lzcnt16(n);
212       }
213   #endif
214   #ifdef CPU_HAS_POP_CNT
215       static unsigned short pop_cnt(T /*unsigned short*/ n) {
216         assert(cpu_has_pop_cnt());
217         return __popcnt16(n);
218       }
219   #endif
220     };
221
222     // Tip: Some CPU's have no support for pop_cnt and lz_cnt.
223     //      Use the following tests in a pre-build step to generate a
224     //      header file with preprocessor definitions for the actual platform.
225     //      Enable features by defining CPU_HAS_LZ_CNT and CPU_HAS_POP_CNT.
226     // Ref: msdn.microsoft.com/en-us/library/hskdteyh(v = VS.90).aspx
227     // Ref: sandpile.org/ia32/cpuid.htm
228
229
230     bool cpu_has_pop_cnt() {
231       // Desc: info_type = 0x00000001 and check bit 23 of cpu_info[2].
232       int cpu_info[4]; const int info_type(0x00000001);
233       __cpuid(cpu_info, info_type);
234       return (cpu_info[2] & (1 << 23)) != 0;
235     }
236     bool cpu_has_lz_cnt() {
237       // Desc: info_type = 0x80000001 and check bit 5 of cpu_info[2].
238       int cpu_info[4]; const int info_type(0x80000001);
239       __cpuid(cpu_info, info_type);
240       return (cpu_info[2] & (1 << 5)) != 0;
241     }
242
243   #elif defined(__GNUC__) // GCC
244
245   // Note: It appears if the GCC intrincics are well defined on any platform.
246   //       Therefore there is no need in general for:
247   //         #if (defined (__amd64__) || defined (__x86_64__)) // 64-bit Intel
248   //           //typedef unsigned long long word_type;
249   //         #elif defined(__i386__) // 32-bit Intel
250   //           //typedef unsigned long word_type;
251   //         #endif
252   // Tip: Compatibility <intrin_x86.h> header for GCC -- GCC equivalents of intrinsic
        Microsoft Visual C++ functions. http://www.koders.com/cpp/
        fidF6A8E722EF8A8D0E3BF173BC25726F66C56BB6B5.aspx
253
254     // 64-bit
255
256     template <typename T>
257     struct bit_man<T,64> : public bit_man_generic<T,64> {
258       static int trailing_zeros(T /*unsigned long long*/ n) {
259         assert(n != 0);
260         return __builtin_ctzll(n);
261       }
262       static int bit_scan_reverse(T /*unsigned long long*/ n) {
263         assert(n != 0);
264         return (sizeof(unsigned long long)*CHAR_BIT) - __builtin_clzll(n) - 1;
265       }
266       static int leading_zeros(T /*unsigned long long*/ n) {
267         assert(n != 0);
268         return __builtin_clzll(n);
269       }
270       static int pop_cnt(T /*unsigned long long*/ n) {
271         return __builtin_popcountll(n);
272       }
273     };
274
275     // 32-bit
276
277     template <typename T>
```

```
278     struct bit_man<T,32> : public bit_man_generic<T,32> {
279       static int trailing_zeros(T /*unsigned long*/ n) {
280         assert(n != 0);
281         return __builtin_ctzl(n);
282       }
283       static int bit_scan_reverse(T /*unsigned long*/ n) {
284         assert(n != 0);
285         return (sizeof(unsigned long)*CHAR_BIT) - __builtin_clzl(n) - 1;
286 
287         // // not tested:
288         // unsigned long index;
289         // __asm__("bsrl %[n], %[index]" : [index] "=r" (index) : [n] "mr" (n));
290         // return index;
291       }
292       static int leading_zeros(T /*unsigned long*/ n) {
293         assert(n != 0);
294         return __builtin_clzl(n);
295       }
296       static int pop_cnt(T /*unsigned long*/ n) {
297         return __builtin_popcountl(n);
298       }
299     };
300 
301     // 16-bit
302 
303     template <typename T>
304     struct bit_man<T,16> : public bit_man_generic<T,16> {
305       static int trailing_zeros(T /*unsigned int*/ n) {
306         assert(n != 0);
307         return __builtin_ctz(n);
308       }
309       static int bit_scan_reverse(T /*unsigned int*/ n) {
310         assert(n != 0);
311         return (sizeof(unsigned int)*CHAR_BIT) - __builtin_clz(n) - 1;
312       }
313       static int leading_zeros(T /*unsigned int*/ n) {
314         assert(n != 0);
315         return __builtin_clz(n);
316       }
317       static int pop_cnt(T /*unsigned int*/ n) {
318         return __builtin_popcount(n);
319       }
320     };
321 
322     #if (defined (__amd64__) || defined (__x86_64__) || defined(__i386__)) // 64-bit or 32-bit
          Intel or AMD
323       static inline int cpuid(int info_type, unsigned long cpu_info[4]) {
324         int highest;
325         asm volatile("cpuid":"=a"(*cpu_info),"=b"(*(cpu_info+1)),
326                      "=c"(*(cpu_info+2)),"=d"(*(cpu_info+3)):"0"(info_type));
327         return highest;
328       }
329     #endif
330 
331 
332   #else
333     #ifdef _MSC_VER
334       #pragma message ( "Using␣naive␣bit␣manipulation,␣please␣implement␣your␣platform␣here!!!" )
335     #else
336       #warning "Using␣naive␣bit␣manipulation,␣please␣implement␣your␣platform␣here!!!"
337     #endif
338   #endif
339 
340     // -----  Traits.
341 
342     // Desc: Smallest bit position data type for indexing a
343     //       pointer type of the size number_of_bits wide.
344     // Use:  typedef bit_position_type : position_bits your_type_name;
345     template <int number_of_bits> struct word_type_traits {
346       typedef word_type_traits<(number_of_bits >> 1)> half_traits;
347       typedef typename half_traits::bit_position_type bit_position_type;
348       static const char position_bits = half_traits::position_bits + 1;
349     };
350     template <> struct word_type_traits<1> {
```

```
351       typedef signed char bit_position_type;
352       static const int position_bits = 0;
353     };
354     template <> struct word_type_traits<(1 << 8)> {
355       typedef word_type_traits<(1 << 7)> half_traits;
356       typedef signed short bit_position_type;
357       static const short position_bits = half_traits::position_bits + 1;
358     };
359     template <> struct word_type_traits<(1 << 16)> {
360       typedef word_type_traits<(1 << 15)> half_traits;
361       typedef signed int bit_position_type;
362       static const int position_bits = half_traits::position_bits + 1;
363     };
364
365     template <int number_of_bits> struct word_traits {
366       typedef word_traits<(number_of_bits - 1)> pred_traits;
367       typedef typename pred_traits::word_type word_type;
368     };
369     template <> struct word_traits<1> {
370       typedef unsigned char word_type;
371     };
372     template <> struct word_traits<9> {
373       typedef unsigned short word_type;
374     };
375     template <> struct word_traits<17> {
376       typedef unsigned long word_type;
377     };
378     template <> struct word_traits<33> {
379       typedef unsigned long long word_type;
380     };
381
382   } // namespace cphstl
383   #endif
```

A.4 bit_manipulation_cpu_config.hpp

```
1   #ifndef _CPHSTL_BIT_MANIPULATION_CPU_CONFIG_
2   #define _CPHSTL_BIT_MANIPULATION_CPU_CONFIG_
3
4   //#define CPU_HAS_LZ_CNT
5   #define CPU_HAS_POP_CNT
6
7   #endif
```

A.5 brown_k_node.hpp

```
1   #ifndef _CPHSTL_PQFW_BROWN_K_NODE_H_
2   #define _CPHSTL_PQFW_BROWN_K_NODE_H_
3
4   /*
5   Desc: Another binomial two-pointer node base.
6   Auth: Asger Bruun 2009-2010
7   Idea: Reversed child ordering, non circular.
8   Ref : M.R. Brown, Implementation and analysis of binomial queue
9         algorithms, SIAM Journal on Computing 7 (1978), 298-319.
10  Note: The binary std::comparator predicate usually does not have the
11        ability to rebind (like the std::allocator). On the other hand
12        we know of usage patterns no other than where a comparator needs
13        to change the value type only and without extra parameters,..
14        and because of this, its quite safe to give our comparator as
15        a "template template parameter", if we like to do that.
16        Bi- and uni-directional root list version are fully defined.
17  */
18
19  #include "assert.h++" // assert
20  #include <cstddef> // std::size_t
21  #include "root_list.hpp" // root_list_node
22  #include "root_list_bidir.hpp"
23  #include "node_with_direct_value.hpp" // value_node
24  #include "type.h++" // if_then_else
25
26  namespace cphstl {
```

```
27    namespace pqfw_node {
28      template <typename D//erived
29      , bool is_const = false>
30      class brown_k_base : public root_list_node<D> {
31      protected:
32        D* c_;
33
34        brown_k_base(D* c) : root_list_node<D>(), c_(c) {}
35      };
36
37      template <typename V = int
38        , typename C = std::less<V>
39        , typename A = std::allocator<V>
40        , bool unidir = true
41      >
42      class brown_k_node : public brown_k_base<brown_k_node<V,C,A,unidir> >, public value_node
            <V,C,A> {
43        typedef brown_k_base<brown_k_node<V,C,A,unidir> > B;
44        friend class brown_k_base<brown_k_node<V,C,A,unidir> >;
45
46        typedef brown_k_node<V,C,A,unidir> N;
47
48        explicit brown_k_node();
49        explicit brown_k_node(N const&);
50        N& operator = (N const&);
51
52      public:
53
54        static const bool promote_is_root_list_neutral = !unidir;
55
56        template<typename V2>
57        struct rebind_value_type {
58          typedef brown_k_node<V2, C, A> other;
59        };
60
61
62        explicit brown_k_node(V const& value) : B(unidir?this:0), value_node<V,C,A>(value) {}
63
64        typedef unsigned char height_type; // assert(size_of(void*) <= (256/8));
65
66        typedef typename cphstl::if_then_else<
67          unidir, root_list<N,false>, root_list_bidir<N>
68        >::type root_list_type;
69
70        typedef brown_k_node node_base_type;
71        typedef typename value_node<V,C,A>::size_type size_type;
72
73        static size_type footprint() { return sizeof(brown_k_node); }
74
75        bool is_valid() const {
76          bool node_is_valid = (*this).B::is_valid()
77            && ((*this).c_ == 0 || B::is_valid((*this).c_));
78          assert(node_is_valid);
79          return(node_is_valid);
80        }
81
82        N* sp() const {
83          assert((*this).is_valid());
84          return (N*) B::next();
85        }
86        N*& sp() {
87          assert((*this).is_valid());
88          return (N*&) B::next();
89        }
90
91        // ----- root list support
92
93        bool prev_rootable() const {
94          assert((*this).is_valid());
95          return (*this).c_ != 0;
96        }
97        N* prev_root() const {
98          assert(!unidir); assert((*this).is_valid());
99          return (*(*this).c_).sp();
```

```
100         }
101         N*& prev_root() {
102           assert(!unidir); assert((*this).is_valid());
103           return (*(*this).c_).sp();
104         }
105
106         // -----
107
108         bool is_root() const {
109           assert((*this).is_valid());
110           if(unidir) {
111             if((*this).c_ == 0) return false;
112             if((*this).c_ == this) return true;
113             return (*(*this).c_).next_ == this;
114           } else return (
115             (*const_cast<N*>(this)).parent() == 0
116             );
117         }
118         bool is_leaf() const {
119           assert((*this).is_valid());
120           return ((*this).c_ == 0 || (unidir && (*this).c_ == this));
121         }
122         bool is_most_significant_child() /*const*/ {
123           assert((*this).is_valid());
124           if((*this).next_ == 0) return false;
125           if(unidir && (*(*this).sp()).c_ == this) return true;
126           if((*(*this).sp()).next_ == 0) return false;
127           return ((*(*(*this).sp()).sp()).c_ == this);
128         }
129
130
131         bool is_marked() const { assert((*this).is_valid()); return false; }
132
133         void swap_roots(N* q) { // replace this root
134           // compared with cph stl restricted to root node replace.
135           assert(this != q);
136           assert(is_root()); assert((*this).next_ == 0);
137           assert((*q).is_root()); assert((*q).is_leaf()); assert((*q).next_ == 0);
138
139           if(is_leaf()) return;
140           std::swap((*this).c_, (*q).c_);
141           if(unidir) (*(*q).c_).sp() = q;
142         }
143
144         N* distinguished_descendant() {
145           assert(0); return 0;  // not implemented
146         }
147
148         N* root() {
149           // Desc: find the root. T(n): O(lg(n)). ES: no throw.
150           assert((*this).is_valid());
151           N* p((N*) this);
152           for(N* q = (*p).parent(); q != 0; q = (*q).parent()) p = q;
153           return p;
154         }
155
156         N* parent() const {
157           // Desc: find parent. T(n): O(lg(n)). ES: no throw.
158           assert((*this).is_valid());
159           N* mss(most_significant_sibling());
160           if((*mss).next_ == 0) return 0;
161           if(unidir && (*(*mss).sp()).c_ == mss) return (*mss).sp();
162           if((*(*mss).sp()).next_ == 0) return 0;
163           if((*(*(*mss).sp()).next()).c_ == mss) return (*(*mss).sp()).next();
164           return 0;
165         }
166
167         N* distinguished_ancestor() {
168           return (*(const_cast<N*>(this))).parent();
169         }
170         N* most_significant_child() const {
171           assert((*this).is_valid());
172           if(unidir && (*this).c_ == this) return 0;
173           if((!unidir) && (*this).c_ != 0 && (*(*this).c_).next_ == this) return 0;
```

```
174           return (*this).c_;
175         }
176         N* most_significant_child() {
177           assert((*this).is_valid());
178           if(unidir && (*this).c_ == this) return 0;
179           if((!unidir) && (*this).c_ != 0 && (*(*this).c_).next_ == this) return 0;
180           return (*this).c_;
181         }
182         N* most_significant_sibling() const /*??*/ {
183           assert((*this).is_valid());
184           N* s(const_cast<N*>(this));
185           for(N* t((*s).more_significant_sibling());
186             t != 0; t = (*t).more_significant_sibling()) s = t;
187           return s;
188         }
189         N* more_significant_sibling() {
190           assert((*this).is_valid());
191           if((*this).next_ == 0) return 0; // last root
192           if(unidir) {
193             if((*this).c_ == this) return (*this).sp(); // single root
194             if((*(*this).sp()).c_ == 0) return 0; // deepest leaf
195             if((*(*this).sp()).c_ == this) return 0; // most signif child of root
196             if((*(*(*this).sp()).c_).next_ == this) return (*this).sp();
197             if((*(*this).c_).next_ == this) return (*this).sp(); // root
198             return 0;
199           } else {
200             if((*(*this).next_).next_ == 0) return (*this).sp(); // second last root
201             if((*this).c_ == 0) {
202               if((*(*(*this).sp()).next()).c_ == this) return 0; // deepest leaf
203               if((*(*this).sp()).c_ == 0) return (*this).sp(); // first of two single roots
204               if((*(*(*this).sp()).c_).sp() == this) return (*this).sp(); // leaf
205               assert(0);
206             }
207             if((*(*(*this).sp()).next()).c_ == this) return 0; //  most signif child
208             return (*this).sp();
209           }
210         }
211         N* less_significant_sibling() {
212           assert((*this).is_valid());
213           if((*this).c_ == 0) return 0;
214           if(unidir) {
215             if((*this).c_ == this) return 0; // root 0
216             if((*(*this).c_).next_ == this) return 0; // root 1
217             else return (*(*this).c_).sp();
218           } else {
219             return (*(*this).c_).sp();
220           }
221         }
222
223         N * successor() const {
224           // T(n): != O(1). ES: no throw.
225           assert((*this).is_valid());
226           if((*this).c_ != 0 && (*this).c_ != this && unidir) return (*this).c_;
227           if((*this).c_ != 0 && (*(*this).c_).next() != this && !unidir) return (*this).c_;
228           N* p((*this).parent());
229           if(p == 0) return (*this).next();
230           N* pp((*p).parent());
231           if(pp == 0) return (*p).next();
232           else return (*p).less_significant_sibling();
233         }
234
235
236         height_type height() const {
237           // Desc: Slow height, usefull for assertions. ES: no throw.
238           assert((*this).is_valid());
239           height_type h(0);
240           for(N* c((*this).most_significant_child()); c != 0
241             ; c = (*c).most_significant_child()) { ++h; }
242           return h;
243         }
244
245         N* join(N* weaker) {
246           // Desc: join with a weaker tree. T(n): O(1). ES: no throw.
247           assert((*this).is_valid()); assert(this != weaker);
```

```
248           assert(!careful || is_root());
249           assert(!careful || (*weaker).is_root());
250           assert((*this).height() == (*weaker).height());
251
252           if(unidir) {
253             if((*this).c_ == this) {
254               assert((*weaker).c_ == weaker);
255               (*weaker).sp() = this;
256               (*this).c_ = weaker;
257               (*weaker).c_ = 0;
258             } else {
259               (*(*weaker).c_).next_ = (*this).c_;
260               (*weaker).next_ = this;
261               (*(*this).c_).next_ = weaker;
262               (*this).c_ = weaker;
263             }
264           } else {
265             if((*this).c_ == 0) {
266               assert((*weaker).c_ == 0);
267               assert((*weaker).sp() == 0);
268               (*this).c_ = weaker;
269             } else {
270               (*(*weaker).c_).next_ = (*this).c_;
271               (*weaker).next_ = (*(*this).c_).next_;
272               (*(*this).c_).next_ = weaker;
273               (*this).c_ = weaker;
274             }
275           }
276           return this;
277         }
278         template <typename C2>
279         N* join(N* other, C2 const& comparator) {
280           if (!comparator((*this).element(), (*other).element()))
281             return join(other);
282           else return (*other).join(this);
283         }
284
285         N* split() {
286           // Desc: split of weaker tree. T(n): O(1). ES: no throw.
287           assert((*this).is_valid()); assert(is_root());
288           assert(most_significant_child() != 0);
289           N* b = (*this).most_significant_child();
290           if(unidir) {
291             //if((*b).is_leaf()) {
292             if((*b).c_ == 0) {
293               (*this).c_ = this;
294               (*b).c_ = b;
295             } else {
296               (*this).c_ = (*(*b).c_).sp();
297               (*(*this).c_).next_ = this;
298               (*(*b).c_).next_ = b;
299             }
300           } else {
301             //if((*b).is_leaf()) {
302             if((*b).c_ == 0) {
303               (*this).c_ = 0;
304               (*b).c_ = 0;
305             } else {
306               (*this).c_ = (*(*b).c_).sp();
307               (*(*this).c_).next_ = (*b).next_;
308               (*(*b).c_).next_ = 0;
309             }
310           }
311           (*b).sp() = 0;
312           return b;
313         }
314
315         template <class Boolean>
316         N* construct(size_type& n, const Boolean calc_size) {
317           // Desc: Vuillemin "construct". T(n): O(1). ES: no throw.
318           //       Isolates the root from its childs.
319           // Ret:  A root stack + total number of nodes incl. the root.
320           assert(!careful || is_root());
321           //if(is_leaf()) { n = 1; return 0; }
```

```
322           if((unidir && (*this).c_ == this) || ((!unidir) && (*this).c_ == 0)) {
323             n = 1; return 0;
324           }
325           N* c((*this).c_);
326           (*(*this).c_).next_ = 0;
327           n = 2;
328           // scan to last root, optionally calc size and/or mark roots.
329           for(N* p((*c).less_significant_sibling()); p != 0; p = (*c).less_significant_sibling()
                  ) {
330             if(unidir) (*(*c).c_).next_ = c;
331             if(calc_size) n <<= 1;
332             c = p;
333           }
334           if(unidir) (*c).c_ = c;
335           (*this).c_ = unidir?this:0;
336           return c;
337         }
338
339         struct hole_type {
340           bool is_leaf;
341           N* da;
342           N* s_out;
343           N* p_out;
344           N* s_in;
345           N* c_in;
346
347           hole_type() : is_leaf(false), da(0)
348             , s_out(0), p_out(0), s_in(0), c_in(0) {}
349           hole_type(N* n, bool auto_splice_out, bool forced = false) :
350             is_leaf((*n).c_ == 0 || (unidir && (*n).c_ == (*n).c_))
351             , da((*n).distinguished_ancestor())
352             , s_out((*n).more_significant_sibling())
353             , p_out(s_out == 0?(*n).sp():0)
354             , s_in((*n).less_significant_sibling()) // equ zero if first single root
355             , c_in(s_out == 0?(*n).parent():0)
356           {
357             assert(!((s_out != 0) && (p_out != 0)));
358             if((auto_splice_out && do_splice()) || forced) {
359               assert((*n).is_valid());
360               (*n).next_ = 0;
361               if(unidir) {
362                 if((*n).c_ == 0) (*n).c_ = n;
363                 else (*(*n).c_).next_ = n;
364               } else {
365                 if((*n).c_ != 0) (*(*n).c_).next_ = 0;
366               }
367               assert((*n).is_root());
368             }
369           }
370           bool is_root() const { return da == 0; }
371           bool is_leaf_root() const { return is_root() && is_leaf; }
372           bool do_splice() {
373             return (unidir && !is_root()) || ((!unidir) && !is_leaf_root());
374           }
375         };
376         hole_type splice_out(bool forced = false) {
377           hole_type hole(this, true,forced);
378           return hole;
379         }
380         template <typename H>
381         void splice_in(hole_type& hole, H& /*heap_store*/,bool forced = false) {
382           if(!(hole.do_splice() || forced)) return;
383           assert((*this).is_root());
384           assert((*this).next_ == 0);
385           if(hole.s_out != 0) (*this).next_ = hole.s_out;
386           if(hole.p_out != 0) (*this).next_ = hole.p_out;
387           if(hole.s_in != 0) (*hole.s_in).next_ = this;
388           if(hole.c_in != 0) (*hole.c_in).c_ = this;
389
390           N* p_in(hole.s_out == 0?0:(*hole.s_out).c_);
391           if(p_in != 0) (*p_in).next_ = this;
392
393           if(unidir) {
394             if((*this).c_ == this) (*this).c_ = 0;
```

```
395             if((*this).c_ != 0) {
396               (*(*this).c_).next_ = hole.s_in;
397             }
398           } else {
399             assert((*this).c_ != this); // !: ensure no mixup with unidir
400             if((*this).c_ == 0) {
401             } else {
402               (*(*this).c_).next_ = hole.s_in;
403             }
404
405             if((*this).c_ != 0) {
406               (*(*this).c_).next_ = hole.s_in;
407             }
408           }
409         }
410
411         N* promote_alt(N* p) {
412           // Vuillemin compatible.
413           promote(p);
414           return (*this).distinguished_ancestor();
415         }
416
417         N* promote(N* p) {
418           // CPH STL compatible.
419           assert((*this).is_valid()); assert((*p).is_valid());
420           assert((*this).distinguished_ancestor() == p);
421
422           N* p_par((*p).parent());
423           bool p_was_root(p_par == 0);
424
425           N* p_lss((*p).less_significant_sibling());
426           N* p_mss((*p).more_significant_sibling());
427           N* p_downto((p_par != 0 && (*p_par).c_ == p)?p_par:0);
428           N* t_lss((*this).less_significant_sibling());
429           N* t_mss((*this).more_significant_sibling());
430           N* t_upto(t_mss != 0?(*t_mss).c_:0);
431
432           if((*p).c_ == this) {
433             if(unidir && p_was_root) {
434               (*this).next_ = (*p).next_;
435               (*p).next_ = this;
436             } else {
437               std::swap((*this).next_,(*p).next_);
438               if(p_mss != 0) (*(*p_mss).c_).next_ = this;
439             }
440             (*p).c_ = (*this).c_;
441             (*this).c_ = p;
442           } else {
443             if(unidir && p_was_root) (*(*p).c_).next_ = this;
444             else if(p_mss != 0) (*(*p_mss).c_).next_ = this;
445             std::swap((*this).next_,(*p).next_);
446             std::swap((*this).c_,(*p).c_);
447           }
448           if(p_downto != 0) (*p_downto).c_ = this;
449           if(t_lss != 0) (*t_lss).next_ = p;
450           if(p_lss != 0) (*p_lss).next_ = this;
451           if(t_upto != 0) (*t_upto).next_ = p;
452
453           return this;
454         }
455
456         std::string str() const { return (*this).str<N>(); }
457
458       protected:
459
460         template <typename D>
461         std::string str() const {
462           std::ostringstream os;
463           if(this == 0) os << "nil";
464           else {
465             os << (*this).element();
466             N const*const x = (*const_cast<N*>(this)).most_significant_child();
467             if(x != 0) {
468               os << "(" << (*static_cast<D const*const>(x)).str();
```

```
469               for(N* y = (*const_cast<N*>(x)).less_significant_sibling();
470                 y != 0; y = (*y).less_significant_sibling())
471                 os << "," << (*static_cast<D *const>(y)).str();
472               os << ")";
473             }
474           }
475           return os.str();
476         }
477       };
478
479       template <typename V, typename C, typename A, bool unidir>
480       struct node_traits<brown_k_node<V,C,A,unidir> >  { enum { has_splice = 1 }; enum {
          has_owner = 0 }; enum { is_root_listable = 1 }; };
481   }
482 }
483 #endif
```

A.6 brown_r_node.hpp

```
1  #ifndef _CPHSTL_PQFW_BROWN_R_NODE_H_
2  #define _CPHSTL_PQFW_BROWN_R_NODE_H_
3
4  /*
5  Desc: Another binomial two-pointer node base.
6  Auth: Asger Bruun 2009-2010
7  Idea: Reversed childordering, circular.
8  Ref : M.R. Brown, Implementation and analysis of binomial queue
9  :     algorithms, SIAM Journal on Computing 7 (1978), 298-319.
10 Note: Bi- and uni-directional root list version are fully defined.
11
12 *  o
13 *  !
14 *  o--------
15 *  :        \
16 * >o-o--o----o---
17 *    |  :\   :   \
18 *   >o >o-o >o-o--o
19 *         |     |  :\
20 *        >o   >o >o-o
21 *                 |
22 *                >o
23 */
24
25 #include "assert.h++" // assert
26 #include <cstddef> // std::size_t
27 #include "root_list.hpp" // root_list_node
28 #include "root_list_bidir.hpp"
29 #include "node_with_direct_value.hpp" // value_node
30
31 namespace cphstl {
32   namespace pqfw_node {
33
34     template <typename D/*erived*/> //, bool is_const = false>
35     class brown_r_base : public root_list_node<D> {
36     protected:
37       D* cp_;
38
39       //typedef root_list_node<D> root_list_base;
40
41       brown_r_base(D* cp) : root_list_node<D>(), cp_(cp) {}
42     };
43
44     template <typename V = int
45       , typename C = std::less<V>
46       , typename A = std::allocator<V>
47       , bool unidir = true
48     >
49     class brown_r_node : public brown_r_base<brown_r_node<V,C,A,unidir> >, public value_node
          <V,C,A> {
50       typedef brown_r_base<brown_r_node<V,C,A,unidir> > B;
51       friend class brown_r_base<brown_r_node<V,C,A,unidir> >;
52
53       typedef brown_r_node<V,C,A,unidir> N;
```

```
54
55        brown_r_node(N const&);
56        N& operator = (N const&);
57
58      public:
59
60        static const bool promote_is_root_list_neutral = !unidir;
61
62        template<typename V2>
63        struct rebind_value_type {
64          typedef brown_r_node<V2, C, A> other;
65        };
66
67        brown_r_node(V const& value) : B(unidir?this:0), value_node<V,C,A>(value) {}
68
69
70        typedef unsigned char height_type; // assert(size_of(void*) <= (256/8));
71        typedef typename cphstl::if_then_else<
72          unidir, root_list<N,false>, root_list_bidir<N>
73        >::type root_list_type;
74
75        typedef brown_r_node node_base_type;
76
77        typedef typename value_node<V,C,A>::size_type size_type;
78
79
80        static size_type footprint() { return sizeof(brown_r_node); }
81
82        bool is_valid() const {
83          bool node_is_valid = (*this).B::is_valid()
84            && ((*this).cp_ == 0 || !is_spooky((*this).cp_));
85          if(!unidir) node_is_valid &= (*this).cp_ != this;
86          assert(node_is_valid);
87          return(node_is_valid);
88        }
89
90        N* s() const {
91          assert((*this).is_valid());
92          return (N*) B::next();
93        }
94        N*& s() {
95          assert((*this).is_valid());
96          return (N*&) B::next();
97        }
98
99        template<typename PA // protected access
100       > static void clear_heap(N* h, PA& pa) {
101         // assert(heap is ejected from list)
102         assert(h != 0);
103         N* p(h);
104         for(N* n((*p).cp_);
105           (unidir && n != h) || ((!unidir) && n != 0);
106           n = (*p).cp_)
107         {
108           pa.destroy(p);
109           p = n;
110         }
111         pa.destroy(p);
112       }
113
114       // ----- root list support
115
116       bool prev_rootable() const {
117         assert((*this).is_valid());
118         return (*this).cp_ != 0;
119       }
120       N* prev_root() const {
121         assert(!unidir); assert((*this).is_valid());
122         return (*(*(*this).cp_).s()).cp_;
123       }
124       N*& prev_root() {
125         assert(!unidir); assert((*this).is_valid());
126         return (*(*(*this).cp_).s()).cp_;
127       }
```

```
128
129         // -----
130
131         N* s_prev() {
132           assert((*this).is_valid());
133           if(unidir && is_root()) return 0;
134           if(unidir && (*(*this).cp_).is_root()) return (*(*this).cp_).cp_;
135           return (*(*(*this).cp_).next()).cp_;
136         }
137         N*& c_prev() {
138           // Desc: node having its cp_ pointing to this.
139           assert((*this).is_valid());
140           if(unidir) {
141             if(is_root()) return (*(*this).cp_).s();
142             else if((*(*(*this).s()).cp_).is_root())
143               return (*(*this).s()).cp_;
144             else return (N*&) (*(*(*this).s()).cp_).s();
145           }
146           else return (N*&) (*(*(*this).s()).cp_).next(); // assert(!(is_root && is_leaf));
147         }
148
149         bool is_root() const {
150           assert((*this).is_valid());
151
152           N* p((N*) this);
153           if((*p).s() == p) return false; // non root leaf
154           if(unidir && (*p).cp_ == p) return true; // single root
155           if(unidir && (*(*p).cp_).s() == 0) return false; // leaf of last root
156           if(unidir && (*(*(*p).cp_).s()).cp_ == p) return true; // root
157           if(unidir) return false;
158           if((!unidir) && (*p).cp_ == 0) return true; // single root
159           return ((*p).s() == 0) || (parent() == 0);
160         }
161         bool is_leaf() const {
162           assert((*this).is_valid());
163           N* p((N*) this);
164           if((*p).s() == p) return true; // bottom leaf
165           if(unidir) {
166             if((*p).cp_ == p) return true; // single root
167             if((*(*p).cp_).s() != 0 && (*(*(*p).cp_).s()).cp_ == p) return false; // root
168             if((*p).s() == 0) return false; // last root
169             if((*(*(*p).s()).cp_).s() == (*(*p).s()).cp_) return true;
170             return false;
171           } else {
172             if((*p).cp_ == 0) return true; // single root
173             if((*p).s() == 0) return false; // last root
174             if((*(*p).s()).cp_ != 0 && (*(*(*p).s()).cp_).next() == (*(*p).s()).cp_)
175               return ((*p).parent() != 0);
176             return false;
177           }
178         }
179         bool is_marked() const { assert((*this).is_valid()); return false; }
180
181         // !: be careful, owner is not valid for weak queue, is for root lists only.
182         void* owner() const { assert(0); return (*this).owner_; }
183         void*& owner() { assert(0 ); return (*this).owner_; }
184
185         void swap_roots(N* q) {
186           // compared with cph stl restricted to root node replace.
187           assert(is_root()); assert((*this).s() == 0);
188           assert((*q).is_root()); assert((*q).is_leaf()); assert((*q).s() == 0);
189
190           if(is_leaf()) return;
191           std::swap((*this).cp_, (*q).cp_);
192           (*(*(*q).cp_).s()).cp_ = q;
193           (*this).cp_ = this;
194         }
195
196
197         N* distinguished_descendant() {
198           assert(0); return 0;  // CPH STL special not implemented.
199         }
200
201
```

```
202         N* root() const {
203           // Desc: find the root. T(n): O(lg(n)). ES: no throw.
204           assert((*this).is_valid());
205           N* p((N*) this);
206           for(N* q = (*p).parent(); q != 0; q = (*q).parent()) p = q;
207           return p;
208         }
209         N* parent() const {
210           // Desc: find parent. T(n): O(lg(n)). ES: no throw.
211           // Note: reliable when structure is valid binomial,
212           //       ie. dont use during a splice out.
213           assert((*this).is_valid());
214
215           N* p((N*) this);
216           if((!unidir) && (*p).cp_ == 0) return 0; // single root
217           if(unidir && (*p).cp_ == p) return 0; // single root
218           if(unidir && (*(*p).cp_).s() == 0) return (*p).cp_; // leaf of last root
219           if(unidir && (*p).s() != p && (*(*(*p).cp_).s()).cp_ == p) return 0; // root
220           while(p != 0) {
221             if((*p).s() == 0) return 0; // last root
222             if(unidir) {
223               if((*p).s() == p) {
224                 if((*(*p).cp_).cp_ == p) return (*p).cp_; // bottom leaf of root
225                 return (*(*p).cp_).s(); // bottom leaf of non root
226               }
227               if((*(*(*p).s()).cp_).s() == (*(*p).s()).cp_ ) {
228                 if((*(*(*p).cp_).cp_).s() == p) return (*p).cp_; // leaf of root
229                 else return (*(*p).cp_).s(); // leaf of non root
230               }
231             } else {
232               if((*p).s() == p) return (*(*p).cp_).next(); // bottom leaf
233               if((*(*(*p).s()).cp_).next() == (*(*p).s()).cp_) {
234                 if((*(*p).cp_).next() != (*p).cp_) return (*(*p).cp_).next(); // leaf
235                 else return 0; // redundant root
236               }
237             }
238             p = (*p).s();
239           }
240           return 0;
241         }
242
243         N* distinguished_ancestor() const {
244           return parent();
245         }
246         N* most_significant_child() {
247           assert((*this).is_valid());
248           if(unidir && (*this).cp_ == this) return 0;
249           return (*this).is_leaf()?0:(*this).cp_;
250         }
251         N* most_significant_child() const {
252           assert((*this).is_valid());
253           if(unidir && (*this).cp_ == this) return 0;
254           return (*this).is_leaf()?0:(*this).cp_;
255         }
256
257         N * successor() const {
258           // T(n): != O(1). ES: no throw.
259           assert((*this).is_valid());
260           if(unidir && (*(*this).cp_).is_root()) return (*(*this).cp_).s();
261           if((!unidir) && (*this).cp_ == 0) return (*this).s(); // single root
262           if((!unidir) && (*(*(*this).cp_).next()).is_root()) return (*(*(*this).cp_).next()).
                s();
263           return (*this).cp_;
264         }
265
266         height_type height() const {
267           // Desc: Slow height, usefull for assertions. ES: no throw.
268           assert((*this).is_valid());
269           height_type h(0);
270           for(N* c((*this).most_significant_child()); c != 0
271             ; c = (*c).most_significant_child()) { ++h; }
272           return h;
273         }
274
```

```
275         N* join(N* weaker) {
276           // Desc: join with a weaker tree. T(n): O(1). ES: no throw.
277           assert((*this).is_valid()); assert(this != weaker);
278           assert(!careful || is_root());
279           assert(!careful || (*weaker).is_root());
280           assert((*this).height() == (*weaker).height());
281
282           if(unidir && (*this).cp_ == this) { // prob this
283             assert((*weaker).cp_ == weaker);
284             (*weaker).s() = weaker;
285             (*this).cp_ = weaker;
286             (*weaker).cp_ = this;
287             return this;
288           } else if((!unidir) && (*this).cp_ == 0) {
289             assert((*weaker).cp_ == 0);
290             (*weaker).s() = weaker;
291             (*this).cp_ = weaker;
292             return this;
293           } else {
294             (*weaker).s() = (*(*this).cp_).s();
295             (*(*(*weaker).cp_).s()).cp_ = (*this).cp_;
296             (*(*this).cp_).s() = weaker;
297             (*this).cp_ = weaker;
298           }
299           return this;
300         }
301         template <typename C2>
302         N* join(N* other, C2 const& comparator) {
303           if (!comparator((*this).element(), (*other).element()))
304             return join(other);
305           else return (*other).join(this);
306         }
307
308         N* split() {
309           // Desc: split of weaker tree. T(n): O(1). ES: no throw.
310           assert((*this).is_valid()); assert(is_root());
311           assert(most_significant_child() != 0);
312           N* b = (*this).most_significant_child();
313           if(unidir) {
314             //if((*b).is_leaf()) {
315             if((*b).s() == b) {
316               (*this).cp_ = this;
317               (*b).cp_ = b;
318               (*b).s() = 0;
319             } else {
320               (*this).cp_ = (*(*(*b).cp_).s()).cp_;
321               (*(*this).cp_).s() = (*b).s();
322               (*(*(*b).cp_).s()).cp_ = b;
323               (*b).s() = 0;
324             }
325           } else {
326             //if((*b).is_leaf()) {
327             if((*b).s() == b) {
328               (*this).cp_ = (*b).cp_;
329               (*b).cp_ = 0;
330               (*b).s() = 0;
331             } else {
332               (*this).cp_ = (*(*(*b).cp_).s()).cp_;
333               (*(*this).cp_).s() = (*b).s();
334               (*(*(*b).cp_).s()).cp_ = 0;
335               (*b).s() = 0;
336             }
337           }
338           return b;
339         }
340
341         template <class Boolean>
342         N* construct(size_type& n, const Boolean calc_size) {
343           // Desc: Vuillemin "construct". T(n): O(1). ES: no throw.
344           //       Isolates the root from its childs.
345           // Ret:  A root stack + total number of nodes incl. the root.
346           assert(!careful || is_root());
347           //if(is_leaf()) { n = 1; return 0; }
348           if((unidir && (*this).cp_ == this) || ((!unidir) && (*this).cp_ == 0)) {
```

```
349             n = 1; return 0;
350           }
351           N* ret((*(*this).cp_).s());
352           (*(*this).cp_).s() = 0;
353
354           (*this).cp_ = unidir?this:(*ret).cp_;
355           (*ret).cp_ = unidir?ret:0;
356           if(calc_size) n = 2;
357           if(calc_size || unidir)
358             for(N* t((*ret).s()); t != 0; t = (*t).s()) {
359               if(unidir) (*(*(*t).cp_).s()).cp_ = t;
360               n <<= 1;
361             }
362           return ret;
363         }
364
365         struct hole_type {
366           static N* special() { const N* nil(0); return const_cast<N*>(nil)-1; }
367           bool is_leaf;
368           N* da;
369           N* s_in;
370           N* s_out;
371           N* cp_in;
372           N* cp_out;
373
374           hole_type() : is_leaf(false)
375             , da(0), s_in(0), s_out(0), cp_in(0), cp_out(0) {}
376           hole_type(N* n, bool auto_splice_out, bool forced = false) :
377             is_leaf((*n).is_leaf())
378             , da((*n).distinguished_ancestor())
379             , s_in((*n).s() != n?((*n).cp_ == 0?0:(*n).s_prev()):special())
380             , s_out((*n).s() != n?(*n).s():special())
381             , cp_in(unidir?(*n).c_prev() : ((*n).s() == 0?0:(*n).c_prev()))
382             , cp_out(is_leaf?(*n).cp_:((*n).cp_ == 0?0:(*(*(*n).cp_).s()).cp_))
383           {
384
385             if((auto_splice_out && do_splice()) || forced) {
386               assert((*n).is_valid());
387               (*n).s() = 0;
388               if(is_leaf) (*n).cp_ = unidir?n:0;
389               else (*(*(*n).cp_).s()).cp_ = unidir?n:0;
390               assert((*n).is_root());
391             }
392           }
393           bool is_leaf_root() { return s_in == 0; }
394           bool is_root() { return da == 0; }
395           bool do_splice() {
396             return !((unidir && is_root()) || ((!unidir) && is_leaf_root()));
397           }
398         };
399         hole_type splice_out(bool forced = false) {
400           hole_type hole(this, true, forced);
401           return hole;
402         }
403         template <typename H>
404         void splice_in(hole_type& hole, H& /*heap_store*/, bool forced = false) {
405           if(!(hole.do_splice() || forced)) return;
406           assert((*this).is_root());
407           assert((*this).s() == 0);
408
409           if(hole.cp_in != 0) (*hole.cp_in).cp_ = this;
410
411           if(hole.s_in == hole_type::special()) { // bottom leaf
412             assert(hole.s_out == hole_type::special());
413             (*this).s() = this;
414             (*this).cp_ = hole.cp_out;
415           } else {
416             if(hole.s_in != 0) (*hole.s_in).next_ = this; // (could be list head)
417             (*this).s() = hole.s_out;
418             if(hole.cp_out != 0) {
419               if(hole.is_leaf) (*this).cp_ = hole.cp_out;
420               else (*(*(*this).cp_).s()).cp_ = hole.cp_out;
421             }
422           }
```

```
423         }
424
425         N* promote_alt(N* p) {
426           // Vuillemin compatible.
427           assert((*this).is_valid()); assert((*p).is_valid());
428           assert((*this).distinguished_ancestor() == p);
429           N* r((*p).distinguished_ancestor());
430           N* t(this);
431
432           N* p_s_in = (unidir && (r == 0))?0:
433             (unidir && (*p).cp_ == r)?(*(*p).cp_).cp_
434             : (*(*(*p).cp_).s()).cp_;
435
436           N* p_cp_in;
437           if(unidir) p_cp_in = (r == 0)?(*(*p).cp_).s()
438             : (((*r).cp_ == p)?r:(*(*(*p).s()).cp_).s());
439           else p_cp_in = (r == 0)?
440             (((*p).s() == 0)?0:(*(*(*p).s()).cp_).s())
441             : (((*r).cp_ == p)?r:(*(*(*p).s()).cp_).s());
442
443           N* t_s_in = (unidir && (*t).cp_ == p)?(*(*t).cp_).cp_ :  (*(*(*t).cp_).next()).cp_;
444           N* t_cp_in = ((*p).cp_ == t)?p:(*(*(*t).s()).cp_).s();
445
446           if(p_s_in != 0) (*p_s_in).next() = this;
447
448           if(t_s_in == this) { // leaf
449             (*this).s() = (*p).s();
450             (*p).s() = p;
451           } else {
452             (*t_s_in).s() = p;
453             std::swap((*this).s(), (*p).s());
454           }
455
456           if((*p).cp_ == this) {
457             if((*this).cp_ == p) {
458               //please dont!! std::swap((*this).cp_, (*p).cp_);
459             } else {
460               (*p).cp_ = (*this).cp_;
461               (*this).cp_ = p;
462               if(p_cp_in != 0) (*p_cp_in).cp_ = this;
463             }
464           } else {
465             if((*this).cp_ == p) {
466               (*this).cp_ = (*p).cp_;
467               (*p).cp_ = this;
468               (*t_cp_in).cp_ = p;
469             } else {
470               std::swap((*this).cp_, (*p).cp_);
471               if(p_cp_in)
472                 std::swap((*t_cp_in).cp_, (*p_cp_in).cp_);
473               else
474                 (*t_cp_in).cp_ = p;
475             }
476           }
477
478           return r;
479         }
480
481         N* promote(N* p) {
482           // CPH STL compatible.
483           assert((*this).is_valid()); assert((*p).is_valid());
484           assert((*this).distinguished_ancestor() == p);
485           N* p_prev((*p).s_prev());
486           N* p_pc((*p).c_prev());
487           N* t_prev((*this).s_prev());
488           N* t_pc((*this).c_prev());
489
490           if(p_prev != 0) (*p_prev).s() = this;
491
492           if(t_prev == this) { // leaf
493             (*this).s() = (*p).s();
494             (*p).s() = p;
495           } else {
496             (*t_prev).s() = p;
```

```
497             std::swap((*this).s(), (*p).s());
498           }
499
500           if((*p).cp_ == this) {
501             if((*this).cp_ == p) {
502               //please dont!! std::swap((*this).cp_, (*p).cp_);
503             } else {
504               (*p).cp_ = (*this).cp_;
505               (*this).cp_ = p;
506               (*p_pc).cp_ = this;
507             }
508           } else {
509             if((*this).cp_ == p) {
510               (*this).cp_ = (*p).cp_;
511               (*p).cp_ = this;
512               (*t_pc).cp_ = p;
513             } else {
514               std::swap((*this).cp_, (*p).cp_);
515               std::swap((*t_pc).cp_, (*p_pc).cp_);
516             }
517           }
518           return this;
519         }
520
521         std::string str() const { return (*this).str<N>(); }
522
523       protected:
524
525         template <typename D>
526         std::string str() const {
527           std::ostringstream os;
528           if(this == 0) os << "nil";
529           else {
530             os << (*this).element();
531             N* x = (*this).most_significant_child();
532             if(x != 0) {
533               x = (*x).s(); // least significant
534               os << "(" << (*static_cast<D *const>(x)).str();
535               for(N* y = (*x).s(); y != x; y = (*y).s())
536                 os << "," << (*static_cast<D *const>(y)).str();
537               os << ")";
538             }
539           }
540           return os.str();
541         }
542       };
543
544       template <typename V, typename C, typename A, bool unidir>
545       struct node_traits<brown_r_node<V,C,A,unidir> >  { enum { has_splice = 1 }; enum {
              has_owner = 0 }; enum { is_root_listable = 1 }; };
546
547     }
548   }
549   #endif
```

A.7 direct_heap_store.hpp

```
1   #ifndef _CPHSTL_DIRECT_HEAP_STORE_H_
2   #define _CPHSTL_DIRECT_HEAP_STORE_H_
3
4   /*
5   Desc: Vuillemin style direct heap store based on binary number system.
6         This old thing appears to pop and extract very fast.
7   Auth: Asger Bruun 2009-2010.
8   Req:  Encapsulator interface with:
9           join, promote, element, construct & next_root.
10        If debugging, then a slow height needs to be supplied too.
11  Tip:  The owner property of the encapsulator may be used as next-root,
12        this choice is left to the tree node implementor.
13  Note: The PQFW_dummy_compliance may safely be removed,
14        if not importing standard node types from the cphstl-PQFW.
15        The allo_ isnt used from here but the
16        pq-frameworks require its presence.
```

```
17
18  Ref:  Jean Vuillemin. A data structure for manipulating priority queues.
19  :     Communications of the ACM 21 (1978), 309-315.
20  */
21
22  #define PQFW_dummy_compliance
23
24  #include <limits> // numeric_limits<size_type>::max()
25  #include "pwh_node.hpp"
26  #include "node_with_direct_value.hpp"
27  #include "node_with_facade.hpp"
28  #include "binary_number_system.hpp"
29
30  #ifndef _MSC_VER
31  namespace cphstl {
32      // GCC does yet not support template type arguments for friend.
33      template <typename VCAEZ_CFG>
34      class vuillemin_extract;
35      template <typename VCAEZ_CFG>
36      class cph_stl_extract;
37      template <typename VCAEZ_CFG>
38      class cormen_extract;
39  }
40  #endif
41
42  #include "heap_store_with_vuillemin_extract.hpp"
43  #include "heap_store_with_cphstl_extract.hpp"
44  #include "heap_store_with_cormen_extract.hpp"
45
46  #include "type.h++" // if_then_else
47
48
49  namespace cphstl {
50    template <typename VCAEZ_CFG>
51    struct meld_using_number_system;
52
53    template <typename VCAEZ_CFG>
54    class direct_heap_store_alt
55      : private VCAEZ_CFG::Z
56    {
57
58      typedef direct_heap_store_alt this_type;
59      typedef typename VCAEZ_CFG::E E;
60      typedef typename E::value_type V;
61      typedef typename E::comparator_type C;
62      //typedef meld_using_number_system<VCAEZ_CFG> M;
63      typedef typename VCAEZ_CFG::M M; // eg: meld_using_number_system<VCAEZ_CFG>
64      typedef typename VCAEZ_CFG::X X; // eg: vuillemin_extract<VCAEZ_CFG>
65
66  #ifdef _MSC_VER
67      friend M;
68      friend X;
69  #else
70      // GCC does yet not support template type arguments for friend.
71      friend class meld_using_number_system<VCAEZ_CFG>;
72      friend class vuillemin_extract<VCAEZ_CFG>;
73      friend class cph_stl_extract<VCAEZ_CFG>;
74      friend class cormen_extract<VCAEZ_CFG>;
75  #endif
76
77    protected:
78
79      typedef typename VCAEZ_CFG::Z number_system_type;
80      typedef typename number_system_type::fast_number_type fast_number_type;
81
82  #if defined(PQFW_dummy_compliance)
83      // fix that the native PQFW tree node types are
84      // hard wired to its standard indirect heap store.
85      struct dummy_compliance {
86        typedef dummy_compliance encapsulator_type;
87        typedef dummy_compliance heap_proxy_type;
88        typedef dummy_compliance heap_store_type;
89        typedef dummy_compliance mark_store_type;
90
```

```
91        void update(void*const) {;}
92        bool is_valid() { return true; }
93      };
94    public:
95      typedef dummy_compliance heap_proxy_type;
96      typedef dummy_compliance heap_store_type;
97      typedef dummy_compliance mark_store_type;
98  #endif
99
100   public:
101
102     typedef E encapsulator_type;
103     typedef typename E::allocator_type::template rebind<E>::other allocator_type;
104     typedef typename E::value_type value_type;
105     typedef typename number_system_type::comparator_type comparator_type;
106     typedef typename number_system_type::root_list_type root_list_type;
107     typedef typename number_system_type::size_type size_type;
108     typedef V& reference;
109     typedef V const& const_reference;
110
111     enum { number_system = number_system_type::number_system };
112
113
114     /* I guess we dont need it, but in case, then its easy to change inner arguments this
          way:
115
116     template<template <typename T> class U>
117     struct wrap_comparator {
118       // Desc: track or replace current comparator.
119       typedef typename E::template wrap_comparator<U>::other new_base;
120       typedef direct_heap_store_alt<
121         new_base
122         , S
123       > other;
124     };
125
126     template<template <typename T> class U>
127     struct wrap_allocator {
128       // Desc: track or replace current allocator.
129       typedef direct_heap_store_alt<
130         typename E::template wrap_allocator<U>::other, S> other;
131     };
132
133     template<typename A>
134     struct replace_allocator {
135       typedef direct_heap_store_alt<
136         typename E::template replace_allocator<A>::other, S> other;
137     };
138     */
139
140   protected:
141
142     static allocator_type allo_;
143
144     // structors
145
146   private:
147     explicit direct_heap_store_alt(root_list_type list, size_type size)
148       : VCAEZ_CFG::Z(list,size) { assert((*this).is_valid()); }
149
150   public:
151     explicit direct_heap_store_alt() : VCAEZ_CFG::Z(0,0) { assert((*this).is_valid()); }
152
153     //template<typename PA> // protected access
154     //void clone(direct_heap_store_alt const& other, PA& pa) {
155     //  // implementation point for a simple faster copy.
156     //  assert(false);
157     //}
158
159     ~direct_heap_store_alt() {
160       assert((*this).empty());
161     }
162
163     // accessors
```

```
164
165       E* find_top() {
166         assert((*this).is_valid());
167         return static_cast<E*>(root_list_type::find_top(number_system_type::comparator));
168       }
169
170       allocator_type get_allocator() const
171       { return allo_; } // multiple-heap-framework needs this.
172
173       comparator_type get_comparator() const
174       { return number_system_type::comparator; }
175
176       size_type footprint(size_type n) const {
177         return sizeof(this_type) + n*E::footprint();
178       }
179
180       E* top() { return find_top(); }
181
182
183       // Make relevant parts of number system public
184
185       comparator_type get_comparator()  { return number_system_type::comparator; }
186       size_type max_size() const { return number_system_type::max_size(); }
187       bool empty() const { return number_system_type::empty(); }
188       size_type size() const { return number_system_type::size(); }
189       E* begin() const { return static_cast<E*>(number_system_type::begin()); }
190       E* end() const { return static_cast<E*>(number_system_type::end()); }
191       bool is_valid() { return number_system_type::is_valid(); }
192       void swap(this_type& h2) { number_system_type::swap(h2); } // for fast top.
193
194       // modifiers
195
196       template<typename PA // protected access
197       > void clear(PA& pa) {
198         // Effect: removes join schedule work from extract.
199         while (!(*this).empty()) {
200           E* p = (*this).eject();
201           E::clear_heap(p,pa);
202         }
203         // CPH STL clear is slower:
204         // while ((*this).size() != 0) {
205         //   E* p = (*this).extract();
206         //   pa.destroy(p);
207         // }
208       }
209
210       E* swap_top(E* top) {
211         // Desc: avoid extra compare from fast top if extract() return top.
212         assert(!C()((*top).element(), (*find_top()).element()));
213         assert((*this).is_valid());
214         E* p(number_system_type::eject());
215         (*p).swap_roots(top);
216         inject(top);
217         assert((*this).is_valid());
218         return p;
219       }
220
221       void insert(E* p) { assert((*p).is_valid()); number_system_type::increment(p); }
222
223       E* extract() { return number_system_type::decrement(); }
224
225       E* extract_top(E* p) {
226         // Desc: this is an optimisation opportunity when unidir.
227         // use this to test if top is stable: assert((*p).is_root());
228         extract(p);
229         return find_top();
230       }
231
232       void extract(E* p) {
233         return X::extract(this,p);
234       }
235
236       void increase(E* p, V const& v) {
237         // Note: Allocator exception safe, but not compare exception safe.
```

```
238         assert((*this).is_valid());
239         assert(!comparator(v, (*p).element()));
240         //try {
241           (*p).element() = v;
242         //}
243         //catch (...) { throw; } // CPH STL don't use try catch here.
244         E* d((*p).distinguished_ancestor());
245         while(d != 0 && comparator((*d).element(),v)) {
246           d = promote(p, d);
247         }
248         assert((*this).is_valid());
249       }
250
251       void after_increase(E* p, E* d) {
252         // For increase and CPH STL extract(p).
253         assert((*this).is_valid());
254         assert(d == (*p).distinguished_ancestor());
255         while(d != 0 && comparator((*d).element(),(*p).element())) {
256           d = promote(p, d);
257         }
258         assert((*this).is_valid());
259       }
260
261       void show() {
262         std::cout << (*this);
263       }
264
265     protected:
266
267       E* promote(E* p, E* d) {
268         assert((*this).is_valid());
269         assert(!(*p).is_root());
270         E* r((*p).promote_alt(d));
271         assert((*p).is_root() == (r == 0));
272         if(r == 0) promoted_to_root(p,d); // join schedule and root list fix
273         assert((*this).is_valid());
274         return r;
275       }
276
277
278     public:
279
280       void meld(this_type& h2) {
281         M::meld(this, &h2);
282       }
283
284     private:
285
286       friend std::ostream & operator <<
287         (std::ostream & os, this_type const& hs)
288       { return os << static_cast<number_system_type const&>(hs); }
289     };
290
291     template<typename VCAEZ_CFG>
292     typename direct_heap_store_alt<VCAEZ_CFG>::allocator_type
293       direct_heap_store_alt<VCAEZ_CFG>::allo_ = allocator_type();
294
295
296     template <typename VCAEZ_CFG>
297     struct meld_using_number_system {
298       // This is the only meld policy because is much faster than CPH STL.
299
300       typedef typename VCAEZ_CFG::H H;
301
302       static void meld(H* h1, H* h2) {
303         (*h1).add(*h2);
304       }
305     };
306
307
308     // --- Backward compatibility ---
309
310     // Convert old style direct heap store template to new compact format.
311     template<typename E0
```

```
312       , template <typename C2, typename E2> class S
313       , int extractor_policy
314     >
315     struct direct_heap_store_cfg {
316       typedef typename E0::value_type V;
317       typedef typename E0::comparator_type C;
318       typedef typename E0::allocator_type A;
319     private:
320       typedef direct_heap_store_cfg<E0,S,extractor_policy> this_type;
321     public:
322       typedef E0 E;
323       typedef S<typename E::comparator_type,E> Z; // number system
324       typedef meld_using_number_system<this_type> M;
325
326       typedef typename cphstl::if_then_else<
327         extractor_policy == 1, vuillemin_extract<this_type>
328         , typename cphstl::if_then_else<
329           extractor_policy == 2, cph_stl_extract<this_type>
330           , typename cphstl::if_then_else<
331             extractor_policy == 3, cormen_extract<this_type>
332             , int // makes illegal extractor_policy fail.
333           >::type
334         >::type
335       >::type X; // extractor
336
337       typedef direct_heap_store_alt<this_type> H;
338
339       // Optionally add this if to support mpq directly:
340       // typedef as_pqfw<with_fast_top<H> > R;
341       // typedef cphstl::node_iterator<E> I;
342       // typedef cphstl::node_iterator<E, true> J;
343     };
344
345     // Old style direct heap store template.
346     template<typename E0 = w_facade<w_direct_value<pwh_node_base> >
347       , template <typename C2, typename E2> class S0 = binary_number
348       , int extractor_policy = 1
349     >
350     class direct_heap_store
351       : public direct_heap_store_alt<
352           direct_heap_store_cfg<E0,S0,extractor_policy>
353       >
354     {
355     public:
356
357       explicit direct_heap_store()
358         : direct_heap_store_alt<direct_heap_store_cfg<E0,S0,extractor_policy> >() {}
359     };
360
361   }
362   #endif
```

A.8 has_member.hpp

```
1   #ifndef _CPHSTL_HAS_MEMBER_HPP_
2   #define _CPHSTL_HAS_MEMBER_HPP_
3
4   /*
5   * Desc: SFINAE test for any member by name (without signature check!).
6   *       Matches anything: methods, attributes, typedefs and inherited members too.
7   *       The member test is taken from a post by Shuvaev Alexander on the RSDN forum,
8   *       and was later on rewritten by Roman Perepelitsa on comp.lang.c++.moderated.
9   * Work: I discarded the signature checks, repaired the problem with non class types
10  *       ("int" is now supported too), removed the hard wirering and added a
11  *       template test for optional flags and some preprocessor macros for usability.
12  * Note: Testing, for types only, is far more simple and is therefore probably safer
13  *       for your code to depend on, if You wish to support unknown compilers too [4].
14  *       Please read the macro DEFINE_STATIC_TEST_HAS_MEMBER_TYPE.
15  * User: This stuff might be of help for various kinds of static method dispatching.
16  * Auth: Asger Bruun, 2010.
17  * Ref1: Shuvaev Alexander, "SFINAE4 to be continued", Russian Software Developer Network,
18  *       www.rsdn.ru/forum/cpp/2759773.1.aspx (use google translate), 08-12-2007.
19  * Ref2: Roman Perepelitsa, "Check whether two functions are compatible using
```

```
    metaprogramming,
20  *       groups.google.com/group/comp.lang.c++.moderated/msg/e5fbc9305539f699, 29-02-2008.
21  * Ref3: Tyson Whitehead, "g++: Full VS Partial Specialization of Nested Templates",
22  *       www.mail-archive.com/debian-gcc@lists.debian.org/msg12334.html, 01-09-2004.
23  * Ref4: D. Vandevoorde and N. M. Josuttis, "C++ Templates - The Complete Guide",
24  *       Addison-Wesley, ISBN 0201734842, 2003, p. 106-107.
25  */
26
27  #include <type_traits> // is_pod, enable_if
28  #include "type.h++" // if_then_else
29  #include "assert.h++"
30
31
32  #ifdef _MSC_VER
33    #if (_MSC_VER<1600) // less than MS VS2010 has no C++0x "enable_if".
34      namespace std {
35        using namespace tr1; // !: VS2008 feature pack TR1 namespace fix!
36        template<bool B, class T> struct enable_if { typedef T type; };
37        template <class T> struct enable_if<false, T> {};
38      }
39    #endif
40  #else
41    // GCC 4.4.1 has C++0x enable_if in namespace std.
42  #endif
43
44
45  // Simple static type test: "bool cphstl::has_member_type_NAME<T>::value".
46  #define DEFINE_STATIC_TEST_HAS_MEMBER_TYPE(NAME) \
47  namespace cphstl { \
48    template <typename T> /* Vandevoorde and Josuttis */ \
49    class has_member_type_##NAME { \
50      typedef char yes; \
51      typedef struct { char a[2]; } no; \
52      template <typename U> static yes test(typename U::copy_type const*); \
53      template <typename U> static no test(...); \
54    public: \
55      static const bool value = sizeof(test<T>(0)) == sizeof(yes); \
56    }; \
57  }
58
59
60  // More complete static test: "bool cphstl::has_member_NAME<T>::value".
61  #define DEFINE_STATIC_TEST_HAS_MEMBER(NAME) \
62  namespace cphstl { \
63    template <typename Type> /* Shuvaev Alexander */ \
64    class has_member_##NAME { \
65      class dummy_class {}; \
66      class yes { char m; }; \
67      class no { yes m[2]; }; \
68      struct BaseMixin { void NAME(){} }; \
69      struct Base : public cphstl::if_then_else<std::is_class<Type>::value, \
70        Type, dummy_class>::type, public BaseMixin { Base(){} }; \
71      template <typename T, T t> \
72      class Helper {}; \
73      template <typename U> \
74      static no deduce(U*, Helper<void (BaseMixin::*)(), &U::NAME>* = 0); \
75      static yes deduce(...); \
76    public: \
77      static const bool value = sizeof(yes) == sizeof(deduce((Base*)(0))); \
78    }; \
79  }
80
81
82  // Static test: "bool cphstl::NAME<T>::value".
83  #define DEFINE_STATIC_TEST_OPTIONAL_MEMBER_FLAG(NAME) \
84  DEFINE_STATIC_TEST_HAS_MEMBER(NAME) \
85  namespace cphstl { \
86    template <typename A, bool has_member> \
87    struct sel_##NAME { static const bool value = false; }; \
88    template <typename A> \
89    struct sel_##NAME <A,true> { static const bool value = A::NAME; }; \
90    template <typename A> \
91    struct NAME { \
92      static const bool value \
```

```
93      = sel_##NAME <A, has_member_##NAME <A>::value >::value; \
94    }; \
95  }
96
97  // Static test: "bool cphstl::enable_custom_NAME<T>::value".
98  #define DEFINE_STATIC_TEST_ENABLE_CUSTOM(NAME) \
99  DEFINE_STATIC_TEST_OPTIONAL_MEMBER_FLAG(enable_custom_##NAME)
100
101 // Static test: "bool cphstl::disable_custom_NAME<T>::value".
102 #define DEFINE_STATIC_TEST_DISABLE_CUSTOM(NAME) \
103 DEFINE_STATIC_TEST_OPTIONAL_MEMBER_FLAG(disable_custom_##NAME)
104
105
106 // Info: the has_copy_constructor doesnt work safely with GCC.
107 #if _MSC_VER
108 #define EXPERIMENTAL_HAS_COPY_CONSTRUCTOR
109 #endif
110 #ifdef EXPERIMENTAL_HAS_COPY_CONSTRUCTOR
111 namespace cphstl {
112   #ifdef _MSC_VER
113     // Ref1: "Compiler Support for Type Traits",
114     //       msdn.microsoft.com/en-us/library/ms177194(v = VS.100).aspx
115     // Ref2: "Standard C++ Library TR1 Extensions Reference",
116     //       msdn.microsoft.com/en-us/library/bb982198(v = VS.90).aspx
117     //       msdn.microsoft.com/en-us/library/bb982198(v = VS.100).aspx
118     template<class T>
119     struct has_copy_constructor {
120       static const bool value = std::is_pod<T>::value || __has_copy(T);
121     };
122   #else
123     // Note: This doesn't work unless having carefully crafted copy constructors.
124     //       Private copy constructors deliberately disabled by no def will brake it.
125     // Ref:  "Is it possible to determine if a class has a default ctor?",
126     //       groups.google.com/group/comp.lang.c++.moderated/msg/dd11c681df214335
127     template <typename A>
128     struct has_copy_constructor {
129       typedef char no;
130       typedef long yes;
131       static A make_A();
132       static yes check(A);  // this alternative will be selected if A can be copy
          constructed
133       static no check(...);
134       static const bool value = sizeof(check(make_A())) == sizeof(yes);
135     };
136   #endif
137 }
138 #endif
139
140
141 #ifdef UNIT_TEST_HAS_MEMBER
142 #include "assert.h++"
143
144 #include "type.h++" // cphstl::bool2type<value>
145 // alternatively: #include <type_traits> // std::integral_constant<bool,value>
146
147 #define STATIC_ASSERT(condition) { int check[(condition)? 1 : -1]; check[0] = 0; }
148
149 DEFINE_STATIC_TEST_HAS_MEMBER(type_1)         // has_member_type_1<T>::value
150 DEFINE_STATIC_TEST_HAS_MEMBER(attribute_1)    // has_member_attribute_1<T>::value
151 DEFINE_STATIC_TEST_HAS_MEMBER(static_attribute_1)  // has_member_static_attribute_1<T>::
      value
152 DEFINE_STATIC_TEST_HAS_MEMBER(method_1)       // has_member_method_1<T>::value
153 DEFINE_STATIC_TEST_HAS_MEMBER(not_impl_1)     // has_member_not_impl_1<T>::value
154
155 DEFINE_STATIC_TEST_OPTIONAL_MEMBER_FLAG(static_flag_1) // static_flag_1<T>::value
156 DEFINE_STATIC_TEST_ENABLE_CUSTOM(method_1)    // enable_custom_method_1<T>::value
157 DEFINE_STATIC_TEST_DISABLE_CUSTOM(method_2)   // disable_custom_method_2<T>::value
158
159 namespace cphstl {
160   namespace has_member {
161
162     // Desc: classes for basic has_member tests.
163
164     struct A {
```

```
165         static const bool static_flag_1 = true;
166         typedef char type_1;
167       private:
168         int attribute_1;
169         static const int static_attribute_1 = 117;
170         int not_impl_1(); // not impl.
171       };
172
173       template<bool enable_custom = false>
174       struct B {
175         explicit B() {}
176         explicit
177 #ifdef _MSC_VER
178           __declspec(nothrow)
179 #endif
180           B(B const& other) {}
181         static const bool static_flag_1 = false;
182         static const bool enable_custom_method_1 = enable_custom;
183         int method_1() { return 0; }
184       };
185
186       template<bool disable = false>
187       struct C {
188         // not a valid declaration: bool static_flag_1;
189         static const bool disable_custom_method_2 = disable;
190         int method_2() { return 0; }
191       };
192
193
194       // Desc: method dispatch by type overloading.
195
196       template<typename O>
197       struct D : A {
198         O o;
199         int method_1() {
200           bool2type<enable_custom_method_1<O>::value> custom_impl;
201           return method_1(custom_impl);
202         }
203         int method_2() {
204           bool2type<disable_custom_method_2<O>::value> no_custom_impl;
205           return method_2(no_custom_impl);
206         }
207       protected:
208         int method_1(bool2type<false>) { return 1; }
209         int method_1(bool2type<true>) { return o.method_1(); }
210         int method_2(bool2type<false>) { return o.method_2(); }
211         int method_2(bool2type<true>) { return 1; }
212       };
213
214
215       // Desc: method dispatch using enable_if.
216       // Note: i forgot to repair the enable_if based test after some expariments that
              turned out bad.
217 /*
218       template<typename O>
219       struct E : A {
220         O o;
221
222         int method_1(void);
223         int method_2(void);
224       };
225       template<typename O>
226       typename std::enable_if<
227         enable_custom_method_1<O>::value, int
228       >::type E<O>::method_1(void) { return o.method_1(); }
229
230       template<typename O>
231       int E<O>::method_1(typename std::enable_if<
232         !enable_custom_method_1<O>::value, void
233       >::type) { return 1; }
234
235       template<typename O>
236       int E<O>::method_2(typename std::enable_if<
237         disable_custom_method_2<O>::value, void
```

```
238       >::type) { return 1; }
239
240       template<typename O>
241       int E<O>::method_2(typename std::enable_if<
242         !enable_custom_method_1<O>::value, void
243       >::type) { return o.method_2(); }
244   */
245
246       // Desc: method dispatch separated into a helper class.
247
248       template<typename O, bool custom_impl>
249       struct F_helper {
250         static int method_1(O& o) { return 1; }
251         static int method_2(O& o) { return 1; }
252       };
253       template<typename O>
254       struct F_helper<O,true> {
255         static int method_1(O& o) { return o.method_1(); }
256         static int method_2(O& o) { return o.method_2(); }
257       };
258
259       template<typename O>
260       struct F : A {
261         O o;
262         int method_1() {
263           const bool custom_impl = enable_custom_method_1<O>::value;
264           return F_helper<O,custom_impl>::method_1(o);
265         }
266         int method_2() {
267           const bool no_custom_impl = disable_custom_method_2<O>::value;
268           return F_helper<O,!no_custom_impl>::method_2(o);
269         }
270       };
271     }
272   }
273
274   void unit_test_has_member() {
275     using namespace cphstl::has_member;
276
277     // not supported: STATIC_ASSERT(!cphstl::has_member_type_1<7>::value);
278
279     STATIC_ASSERT(!cphstl::has_member_type_1<int>::value);
280     STATIC_ASSERT(!cphstl::has_member_type_1<A*>::value);
281
282     STATIC_ASSERT(cphstl::has_member_type_1<A>::value);
283     STATIC_ASSERT(!cphstl::has_member_type_1<B<> >::value);
284     STATIC_ASSERT(cphstl::has_member_type_1<D<B< > > >::value);
285
286     STATIC_ASSERT(cphstl::has_member_attribute_1<A>::value);
287     STATIC_ASSERT(!cphstl::has_member_attribute_1<B<> >::value);
288     STATIC_ASSERT(cphstl::has_member_attribute_1<D<B< > > >::value);
289
290     STATIC_ASSERT(cphstl::has_member_static_attribute_1<A>::value);
291     STATIC_ASSERT(!cphstl::has_member_static_attribute_1<B<> >::value);
292     STATIC_ASSERT(cphstl::has_member_static_attribute_1<D<B< > > >::value);
293
294     STATIC_ASSERT(!cphstl::has_member_method_1<A>::value);
295     STATIC_ASSERT(cphstl::has_member_method_1<B<> >::value);
296     STATIC_ASSERT(cphstl::has_member_method_1<D<B< > > >::value);
297
298     STATIC_ASSERT(cphstl::has_member_not_impl_1<A>::value);
299     STATIC_ASSERT(!cphstl::has_member_not_impl_1<B<> >::value);
300
301     STATIC_ASSERT(cphstl::static_flag_1<A>::value);
302     STATIC_ASSERT(!cphstl::static_flag_1<B<> >::value);
303     STATIC_ASSERT(!cphstl::static_flag_1<C<> >::value);
304
305     // runtime test of exposed static method dispatch in D:
306     D<A> da; assert(da.method_1() == 1);
307     D<B<true> > dbt; assert(dbt.method_1() == 0);
308     D<B<false> > dbf; assert(dbf.method_1() == 1);
309     D<C<true> > dct; assert(dct.method_2() == 1);
310     D<C<false> > dcf; assert(dcf.method_2() == 0);
311
```

```
312     //// runtime test of hidden static method dispatch in F:
313     //E<A> ea; assert(ea.method_1() == 1);
314     //E<B<true> > ebt; assert(ebt.method_1() == 0);
315     //E<B<false> > ebf; assert(ebf.method_1() == 1);
316     //E<C<true> > ect; assert(ect.method_2() == 1);
317     //E<C<false> > ecf; assert(ecf.method_2() == 0);
318
319     // runtime test of hidden static method dispatch in F:
320     F<A> fa; assert(fa.method_1() == 1);
321     F<B<true> > fbt; assert(fbt.method_1() == 0);
322     F<B<false> > fbf; assert(fbf.method_1() == 1);
323     F<C<true> > fct; assert(fct.method_2() == 1);
324     F<C<false> > fcf; assert(fcf.method_2() == 0);
325
326   #ifdef _MSC_VER
327     // TR1
328     STATIC_ASSERT(std::has_trivial_copy<A>::value);
329     STATIC_ASSERT(std::has_trivial_copy<D<A> >::value);
330     STATIC_ASSERT(!std::has_trivial_copy<B<> >::value);
331     STATIC_ASSERT(!std::has_trivial_copy<D<B<> > >::value);
332
333     STATIC_ASSERT(std::has_nothrow_copy<A>::value);
334     STATIC_ASSERT(std::has_nothrow_copy<D<A> >::value);
335     STATIC_ASSERT(std::has_nothrow_copy<B<> >::value);
336     STATIC_ASSERT(std::has_nothrow_copy<D<B<> > >::value);
337   #endif
338   }
339   #endif
340   #endif
```

A.9 heap_store_as_pqfw.hpp

```
1   #ifndef _CPHSTL_HEAP_STORE_AS_PQFW_H_
2   #define _CPHSTL_HEAP_STORE_AS_PQFW_H_
3
4   /*
5   Desc: Makes the heap_store compatible with the priority-queue-framework.
6   :     The conflicting constructors made a mess before this wrap (pqfw
7   :     uses dynamic comparator and allocator and dhs uses static).
8   Auth: Asger Bruun 2009-2010.
9   */
10  namespace cphstl {
11
12    template <typename B = direct_heap_store<> >
13    class as_pqfw : public B {
14      typedef typename B::encapsulator_type E;
15      typedef typename E::value_type V;
16      typedef typename B::comparator_type C;
17      typedef typename B::allocator_type A;
18
19      as_pqfw(as_pqfw const&);
20
21    public:
22      as_pqfw(C const& /*c = C()*/, A const& /*a = allocator_type()*/) : B() {}
23    };
24  }
25
26  #endif
```

A.10 heap_store_config.hpp

```
1   #ifndef _CPHSTL_PQFW_HEAP_STORE_CONFIG_H_
2   #define _CPHSTL_PQFW_HEAP_STORE_CONFIG_H_
3
4   /*
5   Desc: Heap store type conversion.
6   Auth: Asger Bruun 2009-2010.
7   */
8
9   #define INCL_LEDA
10  #define INCL_PQFWS
11
```

```
12 #include "stl-meldable-priority-queue.h++"
13 #include "multiple-heap-framework.h++"
14 #include "priority-queue-iterator.h++"
15
16 #include "direct_heap_store.hpp"
17 #include "heap_store_with_cormen_extract.hpp"
18 #include "heap_store_with_cphstl_extract.hpp"
19 #include "heap_store_with_fast_top.hpp"
20 #include "node-iterator.h++"
21 #include "node_config.hpp"
22
23 #include "priority_queue_framework_for_dbhs.hpp"
24 #include "heap_store_as_pqfw.hpp"
25
26 #include "redundant_binary_number_system.hpp"
27
28 //#include "perfect-component.h++"
29 #include "heap-proxy.h++"
30
31
32 #ifdef INCL_LEDA
33 #include "heap_store_config_leda.hpp"
34 #endif
35 #ifdef INCL_PQFWS
36 #include "heap_store_config_pqfws.hpp"
37 #endif
38
39
40 namespace cphstl {
41   namespace pqfw_node {
42
43     template <typename E>
44     struct heap_store_bind {
45       // encapsulator -> realisator.
46       typedef typename E::value_type V;
47       typedef typename E::comparator_type C;
48       typedef typename E::allocator_type A;
49
50       typedef direct_heap_store<E> dhs_slow_top;
51       typedef with_fast_top<direct_heap_store<E> > dhs;
52       typedef pqfw_for_dhs_nodes<V, C, A, E> pqfw;
53     };
54
55     template <typename E>
56     struct hs_db_slow_top {
57       typedef typename E::value_type V;
58       typedef typename E::comparator_type C;
59       typedef typename E::allocator_type A;
60       typedef direct_heap_store<E, binary_number, 1> R;
61       //typedef with_cphstl_extract<direct_heap_store<E, binary_number, 1> > R2;
62       typedef direct_heap_store<E, binary_number, 2> R2;
63       typedef typename R::heap_proxy_type P;
64       typedef cphstl::node_iterator<E> I;
65       typedef cphstl::node_iterator<E, true> J;
66       typedef cphstl::meldable_priority_queue<V,C,A,E,as_pqfw<R>,I,J> mpq;
67       typedef cphstl::meldable_priority_queue<V,C,A,E,as_pqfw<R2>,I,J> mpq2;
68     };
69
70     template <typename E>
71     struct hs_db { // direct binary
72       typedef typename E::value_type V;
73       typedef typename E::comparator_type C;
74       typedef typename E::allocator_type A;
75       typedef with_fast_top<direct_heap_store<E> > R;
76       typedef with_fast_top<direct_heap_store<E, binary_number, 3> > R_with_cormen_extract;
77       typedef with_fast_top<direct_heap_store<E, binary_number, 2> > R_with_cphstl_extract;
78       typedef with_fast_top<direct_heap_store<E, binary_number, 2> > R2;
79       typedef typename R::heap_proxy_type P;
80       typedef cphstl::node_iterator<E> I;
81       typedef cphstl::node_iterator<E, true> J;
82       typedef cphstl::meldable_priority_queue<V,C,A,E,as_pqfw<R>,I,J> mpq;
83       typedef cphstl::meldable_priority_queue<V,C,A,E,as_pqfw<R2>,I,J> mpq2;
84     };
85
```

```
86
87      template <typename E>
88      struct hs_drb { // direct redundant binary
89        typedef typename E::value_type V;
90        typedef typename E::comparator_type C;
91        typedef typename E::allocator_type A;
92        typedef with_fast_top<direct_heap_store<E, js_policy<join_schedule_ultralight>::
              redundant_binary_number> > R_js_u;
93        typedef with_fast_top<direct_heap_store<E, js_policy<join_schedule_light>::
              redundant_binary_number> > R_js_l;
94        typedef with_fast_top<direct_heap_store<E, js_policy<join_schedule_medium>::
              redundant_binary_number> > R_js_m;
95        typedef with_fast_top<direct_heap_store<E, js_policy<join_schedule_fat>::
              redundant_binary_number> > R_js_f;
96        typedef with_fast_top<direct_heap_store<E, js_policy<join_schedule_ultralight>::
              redundant_binary_number, 2> > R_js_u2;
97        typedef direct_heap_store<E, js_policy<join_schedule_ultralight>::
              redundant_binary_number, 2> R_js_u2_sl;
98        typedef with_fast_top<direct_heap_store<E, js_policy<join_schedule_fat>::
              redundant_binary_number, 2> > R_js_f2;
99        typedef direct_heap_store<E, js_policy<join_schedule_fat>::redundant_binary_number, 2>
              R_js_f2_sl;
100       typedef with_fast_top<direct_heap_store<E, js_policy<>::redundant_binary_number, 3> >
              R_with_cormen_extract;
101       typedef with_fast_top<direct_heap_store<E, js_policy<>::redundant_binary_number, 2> >
              R_with_cphstl_extract;
102       //typedef typename R::heap_proxy_type P;
103       typedef cphstl::node_iterator<E> I;
104       typedef cphstl::node_iterator<E, true> J;
105       typedef cphstl::meldable_priority_queue<V,C,A,E,as_pqfw<R_js_u>,I,J> mpq_u;
106       typedef cphstl::meldable_priority_queue<V,C,A,E,as_pqfw<R_js_l>,I,J> mpq_l;
107       typedef cphstl::meldable_priority_queue<V,C,A,E,as_pqfw<R_js_m>,I,J> mpq_m;
108       typedef cphstl::meldable_priority_queue<V,C,A,E,as_pqfw<R_js_f>,I,J> mpq_f;
109       typedef cphstl::meldable_priority_queue<V,C,A,E,as_pqfw<R_js_u2>,I,J> mpq_u2;
110       typedef cphstl::meldable_priority_queue<V,C,A,E,as_pqfw<R_js_f2>,I,J> mpq_f2;
111       typedef cphstl::meldable_priority_queue<V,C,A,E,as_pqfw<R_js_u2_sl>,I,J> mpq_u2_sl;
112       typedef cphstl::meldable_priority_queue<V,C,A,E,as_pqfw<R_js_f2_sl>,I,J> mpq_f2_sl;
113     };
114
115     template <typename E>
116     struct hs_drb_slow_top {
117       typedef typename E::value_type V;
118       typedef typename E::comparator_type C;
119       typedef typename E::allocator_type A;
120       typedef direct_heap_store<E, js_policy<>::redundant_binary_number> R;
121       typedef typename R::heap_proxy_type P;
122       typedef cphstl::node_iterator<E> I;
123       typedef cphstl::node_iterator<E, true> J;
124       typedef cphstl::meldable_priority_queue<V,C,A,E,as_pqfw<R>,I,J> mpq;
125     };
126
127     template <typename E>
128     struct hs_ir { // indirect redundant binary
129       // priority_queue_framework -> meldable_priority_queue.
130       typedef typename E::value_type V;
131       typedef typename E::comparator_type C;
132       typedef typename E::allocator_type A;
133       typedef pqfw_for_dhs_nodes<V,C,A,E> R;
134       typedef heap_proxy<E> P;
135       typedef cphstl::priority_queue_iterator<E, R> I;
136       typedef cphstl::priority_queue_iterator<E, R, true> J;
137       typedef cphstl::meldable_priority_queue<V,C,A,E,R,I,J> mpq;
138     };
139
140
141     template <typename DHS>
142     struct dhs {
143       // direct_heap_store to meldable_priority_queue
144       typedef typename DHS::value_type V;
145       typedef typename DHS::comparator_type C;
146       typedef typename DHS::allocator_type A;
147       typedef typename DHS::encapsulator_type E;
148       typedef typename DHS::heap_proxy_type P;
149       typedef DHS R;
```

```
150         typedef cphstl::node_iterator<E> I;
151         typedef cphstl::node_iterator<E, true> J;
152         typedef cphstl::meldable_priority_queue<V,C,A,E,as_pqfw<R>,I,J> mpq;
153       };
154
155       template <typename V = char
156         , typename C = std::greater<V>
157         , typename A = std::allocator<V>
158       >
159       struct vuillemin {
160         template <typename Runner>
161         static void registrate(Runner& r) {
162           typedef node_config<V,C,A> nc;
163
164           // direct binary number system
165           r.registrate<typename hs_db_slow_top<typename nc::bno_3_dv>::mpq>(
166             "db_bno_3_dv_slow_top");
167           r.registrate<typename hs_db<typename nc::bno_3_dv>::mpq>(
168             "db_bno_3_dv");
169           r.registrate<typename hs_db_slow_top<typename nc::pwh_3_dv>::mpq>(
170             "db_pwh_3_dv_slow_top");
171           r.registrate<typename hs_db<typename nc::pwh_3_dv>::mpq>("db_pwh_3_dv"); // ***
                package
172           r.registrate<typename hs_db<typename nc::pwh_3_iv>::mpq>("db_pwh_3_iv");
173           r.registrate<typename hs_db_slow_top<typename nc::pwh_3_dv>::mpq2>(
174             "db_pwh_3_dv2_slow_top");
175           r.registrate<typename hs_db<typename nc::pwh_3_dv>::mpq2>("db_pwh_3_dv2"); // ***
                select
176
177           //r.registrate<typename hs_db<typename nc::pwh_3_iv>::mpq2>("db_pwh_3_iv2");
178           r.registrate<typename hs_drb<typename nc::pwh_3_dv>::mpq_f>("dl(f)_pwh_3_dv"); //
                *** package
179           r.registrate<typename hs_drb<typename nc::pwh_3_dv>::mpq_u>("dl(u)_pwh_3_dv");
180           // r.registrate<typename hs_drb<typename nc::pwh_3_dv>::mpq_l>("dl(l)_pwh_3_dv");
181           r.registrate<typename hs_drb<typename nc::pwh_3_dv>::mpq_m>("dl(m)_pwh_3_dv");
182           r.registrate<typename hs_drb<typename nc::pwh_3_dv>::mpq_f2_sl>("dl(f)
                _pwh_3_dv2_slow_top"); // *** select
183           r.registrate<typename hs_drb<typename nc::pwh_3_dv>::mpq_f2>("dl(f)_pwh_3_dv2"); //
                *** select
184           r.registrate<typename hs_drb<typename nc::pwh_3_dv>::mpq_u2_sl>("dl(u)
                _pwh_3_dv2_slow_top"); // *** select
185           r.registrate<typename hs_drb<typename nc::pwh_3_dv>::mpq_u2>("dl(u)_pwh_3_dv2"); //
                *** select
186
187           r.registrate<typename hs_db<typename nc::bno_2_dv>::mpq>("db_bno_2_dv");
188
189           r.registrate<typename hs_db<typename nc::bno_r_dv_bidir>::mpq>("db_bno_r_dv_bidir");
                // *** select
190           r.registrate<typename hs_db<typename nc::bno_r_dv_bidir>::mpq2>("db_bno_r_dv_bidir2"
                ); // *** select
191           r.registrate<typename hs_drb<typename nc::bno_r_dv_bidir>::mpq_f2_sl>("dl(f)
                _bno_r_dv2_bidir_slow_top"); // *** select
192           r.registrate<typename hs_drb<typename nc::bno_r_dv_bidir>::mpq_f2>("dl(f)
                _bno_r_dv2_bidir"); // *** select
193           r.registrate<typename hs_drb<typename nc::bno_r_dv_bidir>::mpq_u2>("dl(u)
                _bno_r_dv2_bidir"); // *** select
194           r.registrate<typename hs_db<typename nc::bno_r_dv>::mpq>("db_bno_r_dv");
195           r.registrate<typename hs_db<typename nc::bno_r_dv>::mpq2>("db_bno_r_dv2");
196           r.registrate<typename hs_drb<typename nc::bno_r_dv>::mpq_f>("dl(f)_bno_r_dv"); //
                *** select
197           r.registrate<typename hs_drb<typename nc::bno_r_dv>::mpq_u>("dl(u)_bno_r_dv"); //
                *** select
198           r.registrate<typename hs_drb<typename nc::bno_r_dv>::mpq_f2>("dl(f)_bno_r_dv2"); //
                *** select
199           r.registrate<typename hs_drb<typename nc::bno_r_dv>::mpq_u2>("dl(u)_bno_r_dv2"); //
                *** select
200
201           r.registrate<typename hs_db<typename nc::bno_k_dv_bidir>::mpq>("db_bno_k_dv_bidir");
                // *** select
202           r.registrate<typename hs_db<typename nc::bno_k_dv_bidir>::mpq2>("db_bno_k_dv_bidir2"
                ); // *** select
203           r.registrate<typename hs_drb<typename nc::bno_k_dv_bidir>::mpq_f2>("dl(f)
                _bno_k_dv2_bidir"); // *** select
204           r.registrate<typename hs_drb<typename nc::bno_k_dv_bidir>::mpq_u2>("dl(u)
```

```
            _bno_k_dv2_bidir"); // *** select
205         r.registrate<typename hs_db<typename nc::bno_k_dv>::mpq>("db_bno_k_dv");
206         r.registrate<typename hs_db<typename nc::bno_k_dv>::mpq2>("db_bno_k_dv2");
207         r.registrate<typename hs_drb<typename nc::bno_k_dv>::mpq_f>("dl(f)_bno_k_dv"); //
                *** select
208         r.registrate<typename hs_drb<typename nc::bno_k_dv>::mpq_u>("dl(u)_bno_k_dv"); //
                *** select
209         r.registrate<typename hs_drb<typename nc::bno_k_dv>::mpq_f2>("dl(f)_bno_k_dv2"); //
                *** select
210         r.registrate<typename hs_drb<typename nc::bno_k_dv>::mpq_u2>("dl(u)_bno_k_dv2"); //
                *** select
211
212         r.registrate<typename hs_db<typename nc::bno_3_iv>::mpq>("db_bno_3_iv");
213         r.registrate<typename hs_drb<typename nc::bno_3_dv>::mpq_f>("dl(f)_bno_3_dv");
214         r.registrate<typename hs_drb<typename nc::bno_3_dv>::mpq_u>("dl(u)_bno_3_dv");
215         // r.registrate<hs_drb<typename nc::bno_3_dv>::mpq_l>("dl(l)_bno_3_dv");
216         r.registrate<typename hs_drb<typename nc::bno_3_dv>::mpq_m>("dl(m)_bno_3_dv");
217
218
219         // indirect binary redundant number system
220         r.registrate<typename hs_ir<typename nc::pwh_3_dv>::mpq>(
221           "ir_pwh_3_dv"); // std weak queue // *** package
222         r.registrate<typename hs_ir<typename nc::bno_3_dv>::mpq>("ir_bno_3_dv");
223         r.registrate<typename hs_ir<typename nc::pwh_3_iv>::mpq>("ir_pwh_3_iv");
224         r.registrate<typename hs_ir<typename nc::bno_3_iv>::mpq>("ir_bno_3_iv");
225         r.registrate<typename hs_ir<typename nc::bno_3_px>::mpq>("ir_bno_3_proxy");
226       }
227     };
228   }
229 }
230 #endif
```

A.11 heap_store_config_alt.hpp

```
1  #ifndef _CPHSTL_PQFW_HEAP_STORE_CONFIG_ALT_H_
2  #define _CPHSTL_PQFW_HEAP_STORE_CONFIG_ALT_H_
3
4  #include "stl-meldable-priority-queue.h++"
5
6  #include "direct_heap_store.hpp"
7
8  #include "node-iterator.h++"
9  #include "node_config.hpp"
10
11 #include "priority_queue_framework_for_dbhs.hpp"
12 #include "heap_store_as_pqfw.hpp"
13
14 #include "redundant_binary_number_system.hpp"
15
16 #include "benchmarking.hpp" // counting_compare and counting_allocator
17
18 namespace cphstl {
19
20   // Elementary VCA configuration:
21
22   template <
23     typename V0 // value
24     , typename C0 = std::less<V0> // comparator
25     , typename A0 = std::allocator<V0> // allocator
26   >
27   struct std_VCA {
28     // Desc: VCA-config for porting from STL to CPH STL.
29     // Tip:  The advanced user writes his/her own non templates
30     //       to obtain the cleanest compiler messages possible.
31     typedef V0 V;
32     typedef C0 C;
33     typedef A0 A;
34   };
35
36   template <typename V0>
37   struct counting_VCA {
38     // Desc: VCA config specialised for performance combi_counter.
39     typedef V0 V;
```

```
40      typedef benchmarking::counting_compare<std::less<V> > C;
41      typedef benchmarking::counting_allocator<V, std::allocator<V> > A;
42    };
43
44
45    // VCA + heap store + mpq configurations all in ones:
46
47    template <typename VCA_CFG>
48    struct binary_number_PWH {
49    private:
50      typedef binary_number_PWH<VCA_CFG> this_type;
51    public:
52      typedef typename VCA_CFG::V V;
53      typedef typename VCA_CFG::C C;
54      typedef typename VCA_CFG::A A;
55    private:
56      typedef node_config<V,C,A> nc;
57    public:
58      typedef typename nc::pwh_3_dv E;
59      typedef binary_number<C,E> Z; // number system
60      typedef meld_using_number_system<this_type> M;
61      typedef cph_stl_extract<this_type> X; // extractor
62      typedef direct_heap_store_alt<this_type> H;
63      typedef as_pqfw<with_fast_top<H> > R;
64      typedef cphstl::node_iterator<E> I;
65      typedef cphstl::node_iterator<E, true> J;
66    };
67
68    template <typename VCA_CFG>
69    struct binary_number_PWH_VX {
70    private:
71      typedef binary_number_PWH_VX<VCA_CFG> this_type;
72    public:
73      typedef typename VCA_CFG::V V;
74      typedef typename VCA_CFG::C C;
75      typedef typename VCA_CFG::A A;
76    private:
77      typedef node_config<V,C,A> nc;
78    public:
79      typedef typename nc::pwh_3_dv E;
80      typedef binary_number<C,E> Z; // number system
81      typedef meld_using_number_system<this_type> M;
82      typedef vuillemin_extract<this_type> X; // extractor
83      typedef direct_heap_store_alt<this_type> H;
84      typedef as_pqfw<with_fast_top<H> > R;
85      typedef cphstl::node_iterator<E> I;
86      typedef cphstl::node_iterator<E, true> J;
87    };
88
89    template <typename VCA_CFG>
90    struct redundant_number_PWH {
91    private:
92      typedef redundant_number_PWH<VCA_CFG> this_type;
93    public:
94      typedef typename VCA_CFG::V V;
95      typedef typename VCA_CFG::C C;
96      typedef typename VCA_CFG::A A;
97    private:
98      typedef node_config<V,C,A> nc;
99    public:
100     typedef typename nc::pwh_3_dv E;
101     typedef js_policy<join_schedule_fat>::redundant_binary_number<C,E> Z;
102     typedef meld_using_number_system<this_type> M;
103     typedef cph_stl_extract<this_type> X;
104     typedef direct_heap_store_alt<this_type> H;
105     typedef as_pqfw<with_fast_top<H> > R;
106     typedef cphstl::node_iterator<E> I;
107     typedef cphstl::node_iterator<E, true> J;
108   };
109
110   template <typename VCA_CFG>
111   struct redundant_number_PWH_VX {
112   private:
113     typedef redundant_number_PWH_VX<VCA_CFG> this_type;
```

```
114     public:
115       typedef typename VCA_CFG::V V;
116       typedef typename VCA_CFG::C C;
117       typedef typename VCA_CFG::A A;
118     private:
119       typedef node_config<V,C,A> nc;
120     public:
121       typedef typename nc::pwh_3_dv E;
122       typedef js_policy<join_schedule_fat >::redundant_binary_number<C,E> Z;
123       typedef meld_using_number_system<this_type> M;
124       typedef vuillemin_extract<this_type> X;
125       typedef direct_heap_store_alt<this_type> H;
126       typedef as_pqfw<with_fast_top<H> > R;
127       typedef cphstl::node_iterator<E> I;
128       typedef cphstl::node_iterator<E, true> J;
129     };
130
131     template <typename VCA_CFG>
132     struct redundant_number_R {
133     private:
134       typedef redundant_number_R<VCA_CFG> this_type;
135     public:
136       typedef typename VCA_CFG::V V;
137       typedef typename VCA_CFG::C C;
138       typedef typename VCA_CFG::A A;
139     private:
140       typedef node_config<V,C,A> nc;
141     public:
142       typedef typename nc::bno_r_dv_bidir E;
143       typedef js_policy<join_schedule_fat >::redundant_binary_number<C,E> Z;
144       typedef meld_using_number_system<this_type> M;
145       typedef cph_stl_extract<this_type> X;
146       typedef direct_heap_store_alt<this_type> H;
147       typedef as_pqfw<with_fast_top<H> > R;
148       typedef cphstl::node_iterator<E> I;
149       typedef cphstl::node_iterator<E, true> J;
150     };
151
152     template <typename VCA_CFG>
153     struct redundant_number_R_VX {
154     private:
155       typedef redundant_number_R_VX<VCA_CFG> this_type;
156     public:
157       typedef typename VCA_CFG::V V;
158       typedef typename VCA_CFG::C C;
159       typedef typename VCA_CFG::A A;
160     private:
161       typedef node_config<V,C,A> nc;
162     public:
163       typedef typename nc::bno_r_dv_bidir E;
164       typedef js_policy<join_schedule_fat >::redundant_binary_number<C,E> Z;
165       typedef meld_using_number_system<this_type> M;
166       typedef vuillemin_extract<this_type> X;
167       typedef direct_heap_store_alt<this_type> H;
168       typedef as_pqfw<with_fast_top<H> > R;
169       typedef cphstl::node_iterator<E> I;
170       typedef cphstl::node_iterator<E, true> J;
171     };
172
173     template <typename VCA_CFG>
174     struct redundant_number_K {
175     private:
176       typedef redundant_number_K<VCA_CFG> this_type;
177     public:
178       typedef typename VCA_CFG::V V;
179       typedef typename VCA_CFG::C C;
180       typedef typename VCA_CFG::A A;
181     private:
182       typedef node_config<V,C,A> nc;
183     public:
184       typedef typename nc::bno_k_dv_bidir E;
185       typedef js_policy<join_schedule_fat >::redundant_binary_number<C,E> Z;
186       //typedef js_policy<join_schedule_ultralight >::redundant_binary_number<C,E> Z;
187       typedef meld_using_number_system<this_type> M;
```

```
188       typedef cph_stl_extract<this_type> X;
189       typedef direct_heap_store_alt<this_type> H;
190       typedef as_pqfw<with_fast_top<H> > R;
191       typedef cphstl::node_iterator<E> I;
192       typedef cphstl::node_iterator<E, true> J;
193     };
194
195
196     // Type registration for benchmark or what ever with friendly names:
197
198     template <typename VCA_CFG>
199     struct vuillemin_alt1 {
200       template <typename Runner>
201       static void registrate(Runner& r) {
202         // New style versus old style:
203         //   binary_number_PWH = db_pwh_3_dv2
204         r.registrate<meldable_priority_queue_alt<binary_number_PWH<VCA_CFG> > >("
              binary_number_PWH");
205         r.registrate<meldable_priority_queue_alt<binary_number_PWH_VX<VCA_CFG> > >("
              binary_number_PWH_VX");
206         r.registrate<meldable_priority_queue_alt<redundant_number_PWH<VCA_CFG> > >("
              redundant_number_PWH");
207         r.registrate<meldable_priority_queue_alt<redundant_number_PWH_VX<VCA_CFG> > >("
              redundant_number_PWH_VX");
208       }
209     };
210     template <typename VCA_CFG>
211     struct vuillemin_alt2 {
212       template <typename Runner>
213       static void registrate(Runner& r) {
214         // New style versus old style:
215         //   redundant_number_R = dl(f)_bno_r_dv2_bidir
216         //   redundant_number_K = dl(f)_bno_k_dv2_bidir
217         r.registrate<meldable_priority_queue_alt<redundant_number_R<VCA_CFG> > >("
              redundant_number_R");
218         r.registrate<meldable_priority_queue_alt<redundant_number_R_VX<VCA_CFG> > >("
              redundant_number_R_VX");
219         r.registrate<meldable_priority_queue_alt<redundant_number_K<VCA_CFG> > >("
              redundant_number_K");
220       }
221     };
222   }
223
224
225   #endif
```

A.12 heap_store_config_leda.hpp

```
1   #ifndef _CPHSTL_PQFW_HEAP_STORE_CONFIG_LEDA_HPP_
2   #define _CPHSTL_PQFW_HEAP_STORE_CONFIG_LEDA_HPP_
3
4   /*
5   Desc: LEDA Heap store type conversion.
6   Auth: Asger Bruun 2010.
7   */
8
9   #include <LEDA/core/p_queue.h>
10  #include <LEDA/core/impl/f_heap.h> // fibonacci-heap
11  #include <LEDA/core/impl/p_heap.h> // pairing-heap
12  #include <LEDA/system/allocator.h> // leda_allocator<T>, leda::std_memory_mgr.clear();
13
14  #include "type.h++" // cphstl::bool2type
15  #include <type_traits> // is_same, remove_pointer
16
17
18  namespace cphstl {
19    namespace cphstl_leda {
20
21      class empty_type {
22      public:
23        empty_type() {}
24      };
25
```

```
26      template <typename V
27        , typename C // STL adaptable binary predicate
28        , typename R // LEDA priority queue
29      >
30      class meldable_priority_queue_facade {
31      private:
32        R realizator;
33
34      public:
35
36        typedef meldable_priority_queue_facade<V,C,R> this_type;
37
38        enum { number_system = 5 }; // for pathological tests.
39        typedef this_type realizator_type;
40        typedef typename std::remove_pointer<typename R::pq_item>::type encapsulator_type;
41        typedef V value_type;
42        typedef int (*comp_func_type) (V const&, V const&);
43        typedef typename R::item iterator;
44        typedef std::size_t size_type;
45
46        static int compare(V const& a, V const& b) {
47          if(C()(a,b)) return 1;
48          else if(C()(b,a)) return -1;
49          else return 0;
50        }
51
52        void clone(this_type const& other, bool2type<false>) {
53          // !: this is a fix for broken " = " in leda_pairing_heap.
54           for(iterator i(other.realizator.first_item()); i != 0
55            ; i = other.realizator.next_item(i)) {
56            V v(other.prio(i));
57            (*this).push(v);
58          }
59        }
60        void clone(this_type const& other, bool2type<true>) {
61          // !: assignment results in access violation if leda_pairing_heap.
62          realizator = other.realizator;
63        }
64
65        explicit meldable_priority_queue_facade(comp_func_type const& cmp = compare)
66          : realizator(cmp) {
67        }
68        explicit meldable_priority_queue_facade(this_type const& other)
69          : realizator(&other.compare) {
70            const bool hc = std::is_same<leda::p_queue<V, empty_type, leda::f_heap>, R>::value;
71            bool2type<hc> has_copy;
72            (*this).clone(other, has_copy);
73        }
74
75        bool empty() const {
76          return realizator.empty();
77        }
78
79        size_type size() const {
80          return realizator.size();
81        }
82
83        iterator top() {
84          return realizator.find_min();
85        }
86
87        const V& prio(iterator it) const
88        { return realizator.prio(it); }
89
90        iterator push(V const& v) {
91          return realizator.insert(v, empty_type::empty_type());
92        }
93
94        void pop() {
95          realizator.del_min();
96        }
97
98        void erase(iterator p) {
99          realizator.del_item(p);
```

```
100         }
101
102         void increase(iterator p, V const& v) {
103           realizator.decrease_p(p, v);
104         }
105
106         void clear () {
107           realizator.clear();
108         }
109
110         void meld(this_type& h2) {
111           // Is this operation really missing in LEDA?
112           // Note: the windows library fibonacci-heap requires a huge stack for meld.
113           while(!h2.empty()) {
114             iterator i(h2.top());
115             V v(h2.prio(i));
116             (*this).push(v);
117             h2.pop();
118           }
119         }
120       };
121
122       template <typename V = char
123         , typename C = std::greater<V>
124         , typename A = std::allocator<V>
125       >
126       struct leda_heap {
127         typedef leda::p_queue<V, empty_type, leda::f_heap> F;
128         typedef leda::p_queue<V, empty_type, leda::p_heap> P;
129         typedef meldable_priority_queue_facade<V, C, F> mpq_fibonacci;
130         typedef meldable_priority_queue_facade<V, C, P> mpq_pairing;
131
132         template <typename Runner>
133         static void registrate(Runner& r) {
134           r.registrate<mpq_fibonacci>("leda_fibonacci_heap");
135           r.registrate<mpq_pairing>("leda_pairing_heap");
136         }
137       };
138     }
139   }
140   #endif
```

A.13 heap_store_config_pqfws.hpp

```
1   #ifndef _CPHSTL_PQFW_HEAP_STORE_CONFIG_PQFWS_HPP_
2   #define _CPHSTL_PQFW_HEAP_STORE_CONFIG_PQFWS_HPP_
3
4   /*
5   Desc: CPH STL PQFWS Heap store type configuration.
6   Auth: Asger Bruun 2010
7   Note: The CPH STL "empty() returns (size() == 0)" is a design flaw.
8
9   2010-3-13 CPH STL snapshot status:
10  binary-heap: 0. ok
11  first-version: 0. // impossible, is selected by macro def.
12  new-weak-queue: 0. ok
13  pennant-queue: 0. ok
14  pq-db-pwh-3-dv: 2. pq-ir-pwh-3-dv: 2. pq-lc-pwh-3-dv: 2.
15  rank-relaxed-weak-queue: 0. fails at runtime if not NDEBUG
16  run-relaxed-weak-queue: 0. ok
17  weak-heap: 0.  ok
18  weak-queue: 0. ok
19  weak-queue-with-fat-nodes: 2.
20
21  */
22
23  #include "stl-meldable-priority-queue.h++"
24
25  #include "multiple-heap-framework.h++"
26  #include "priority-queue-iterator.h++"
27
28  #include "pennant-node.h++"
29  #include "weak-heap-node.h++"
```

```
30 #include "fat-weak-heap-node.h++"
31
32 #include "blank-mark-store.h++"
33 #include "proxy-array-heap-store.h++"
34 #include "eager-mark-store.h++"
35 #include "proxy-list-heap-store.h++"
36 #include "lazy-mark-store.h++"
37
38 #include "single-heap-framework.h++"
39 #include "element-encapsulator.h++"
40 #include "proxy-iterator.h++"
41 #include "weak-heap-heapifier.h++"
42
43
44 namespace cphstl {
45   // ----- frameworks
46
47   template <
48     typename V,
49     typename C = std::less<V>,
50     typename A = std::allocator<V>,
51     typename E = weak_heap_node<V, A>,
52     typename H = proxy_list_heap_store<C, A, E>,
53     typename M = blank_mark_store<C, A, E>
54   >
55   class mhfw : public multiple_heap_framework<V,C,A,E,H,M> {
56     typedef multiple_heap_framework<V,C,A,E,H,M> B;
57     typedef typename B::size_type size_type;
58
59   public:
60     enum { number_system = 1 };
61     mhfw(
62       C const& c = C(), A const& a = A()
63       ) : multiple_heap_framework<V,C,A,E,H,M>(c,a) {}
64
65     bool empty() const {
66       return (*this).size() == size_type(0);
67     }
68   };
69
70   template <
71     typename V,
72     typename C = std::less<V>,
73     typename A = std::allocator<V>,
74     typename E = element_encapsulator<V, std::ptrdiff_t , A>,
75     typename H = top_down_binary_heap_heapifier,
76     typename K = std::vector<E*, A>,
77     typename S = vector_surrogate<E*, K>
78   >
79   class shfw : public single_heap_framework<V,C,A,E,H,K,S> {
80     typedef single_heap_framework<V,C,A,E,H,K,S> B;
81     typedef typename B::size_type size_type;
82
83   public:
84     enum { number_system = 6 };
85     shfw(
86       C const& c = C(), A const& a = A()
87       ) : single_heap_framework<V,C,A,E,H,K,S>(c,a) {}
88
89     bool empty() const {
90       return (*this).size() == size_type(0);
91     }
92   };
93
94   // ----- queue types
95
96   template <typename V = char
97     , typename C = std::greater<V>
98     , typename A = std::allocator<V>
99   >
100  struct binary_heap {
101    typedef element_encapsulator<V, std::size_t , A> E;
102    typedef top_down_binary_heap_heapifier H;
103    typedef shfw<V,C,A,E,H> R;
```

```
104       typedef proxy_iterator<E, R> I;
105       typedef proxy_iterator<E, R, true> J;
106       typedef meldable_priority_queue<V,C,A,E,R,I,J> mpq;
107     };
108
109     template <typename V = char
110       , typename C = std::greater<V>
111       , typename A = std::allocator<V>
112     >
113     struct pennant_queue {
114       typedef pennant_node<V,A> E;
115       typedef proxy_list_heap_store<C, A, E> H;
116       typedef blank_mark_store<C, A, E> M;
117       typedef mhfw<V,C,A,E,H,M> R;
118       typedef priority_queue_iterator<E, R> I;
119       typedef priority_queue_iterator<E, R, true> J;
120       typedef meldable_priority_queue<V,C,A,E,R,I,J> mpq;
121     };
122
123     template <typename V = char
124       , typename C = std::greater<V>
125       , typename A = std::allocator<V>
126     >
127     struct new_weak_queue {
128       typedef weak_heap_node<V,A> E;
129       typedef proxy_array_heap_store<C, A, E> H;
130       typedef blank_mark_store<C, A, E> M;
131       typedef mhfw<V,C,A,E,H,M> R;
132       typedef priority_queue_iterator<E, R> I;
133       typedef priority_queue_iterator<E, R, true> J;
134       typedef meldable_priority_queue<V,C,A,E,R,I,J> mpq;
135     };
136
137     template <typename V = char
138       , typename C = std::greater<V>
139       , typename A = std::allocator<V>
140     >
141     struct rank_relaxed_weak_queue {
142       typedef fat_weak_heap_node<V,A> E;
143       typedef proxy_list_heap_store<C, A, E> H;
144       typedef eager_mark_store<C, A, E> M;
145       typedef mhfw<V,C,A,E,H,M> R;
146       typedef priority_queue_iterator<E, R> I;
147       typedef priority_queue_iterator<E, R, true> J;
148       typedef meldable_priority_queue<V,C,A,E,R,I,J> mpq;
149     };
150
151     template <typename V = char
152       , typename C = std::greater<V>
153       , typename A = std::allocator<V>
154     >
155     struct run_relaxed_weak_queue {
156       typedef fat_weak_heap_node<V,A> E;
157       typedef proxy_list_heap_store<C, A, E> H;
158       typedef lazy_mark_store<C, A, E> M;
159       typedef mhfw<V,C,A,E,H,M> R;
160       typedef priority_queue_iterator<E, R> I;
161       typedef priority_queue_iterator<E, R, true> J;
162       typedef meldable_priority_queue<V,C,A,E,R,I,J> mpq;
163     };
164
165     template <typename V = char
166       , typename C = std::greater<V>
167       , typename A = std::allocator<V>
168     >
169     struct weak_heap {
170       typedef element_encapsulator<V, std::ptrdiff_t, A> E;
171       typedef weak_heap_heapifier H;
172       typedef shfw<V,C,A,E,H> R;
173       typedef proxy_iterator<E, R> I;
174       typedef proxy_iterator<E, R, true> J;
175       typedef meldable_priority_queue<V,C,A,E,R,I,J> mpq;
176     };
177
```

```
178     template <typename V = char
179       , typename C = std::greater<V>
180       , typename A = std::allocator<V>
181     >
182     struct weak_queue {
183       typedef weak_heap_node<V,A> E;
184       typedef proxy_list_heap_store<C,A,E> H;
185       typedef blank_mark_store<C,A,E> M;
186       typedef mhfw<V,C,A,E,H,M> R;
187       typedef priority_queue_iterator<E, R> I;
188       typedef priority_queue_iterator<E, R, true> J;
189       typedef meldable_priority_queue<V,C,A,E,R,I,J> mpq;
190     };
191
192     template <typename V = char
193       , typename C = std::greater<V>
194       , typename A = std::allocator<V>
195     >
196     struct pqfws {
197       template <typename Runner>
198       static void registrate(Runner& r) {
199         r.registrate<typename binary_heap<V,C,A>::mpq>("binary_heap");
200         r.registrate<typename pennant_queue<V,C,A>::mpq>("pennant_queue");
201         r.registrate<typename new_weak_queue<V,C,A>::mpq>("new_weak_queue");
202         #ifdef NDEBUG // rank_relaxed_weak_queue fails when debug
203           r.registrate<typename rank_relaxed_weak_queue<V,C,A>::mpq>("rank_relaxed_weak_queue")
                ;
204         #endif
205         // !!!: fails stefan size 10: r.registrate<typename run_relaxed_weak_queue<V,C,A>::
              mpq>("run_relaxed_weak_queue");
206         r.registrate<typename weak_heap<V,C,A>::mpq>("weak_heap");
207         r.registrate<typename weak_queue<V,C,A>::mpq>("weak_queue");
208       }
209     };
210   }
211   #endif
```

A.14 heap_store_with_cormen_extract.hpp

```
1   #ifndef _CPHSTL_HEAP_STORE_WITH_CORMEN_EXTRACT_H_
2   #define _CPHSTL_HEAP_STORE_WITH_CORMEN_EXTRACT_H_
3
4   /*
5   Desc: The Cormen algorithms textbook extract(p) policy.
6   Auth: Asger Bruun 2009-2010.
7   */
8
9   #include "direct_heap_store.hpp"
10
11  namespace cphstl {
12
13    template <typename VCAEZ_CFG>
14    struct cormen_extract {
15      typedef typename VCAEZ_CFG::E E;
16      typedef typename VCAEZ_CFG::Z Z;
17      typedef typename VCAEZ_CFG::H H;
18      typedef typename Z::size_type size_type;
19      typedef typename Z::fast_number_type fast_number_type;
20
21      static void extract(H* hs, E* p) {
22        assert((*hs).is_valid());
23        for(E* d((*p).distinguished_ancestor());
24          d != 0; d = (*p).distinguished_ancestor()
25          ) { (*hs).promote(p, d); }
26        assert((*p).is_root());
27        assert((*hs).is_valid());
28        size_type n((*hs).remove_root(p));
29        if((*p).is_leaf()) {
30          assert((*hs).is_valid());
31        } else {
32          size_type dummy;
33          E* r((*p).construct(dummy, constant<bool,false>()));
34          fast_number_type hs2(r, n-1);
```

```
35          (*hs).add(hs2);
36        }
37        assert((*p).is_root());
38        assert((*p).height() == 0);
39      }
40    };
41  }
42  #endif
```

A.15 heap_store_with_cphstl_extract.hpp

```
1   #ifndef _CPHSTL_HEAP_STORE_WITH_CPHSTL_EXTRACT_H_
2   #define _CPHSTL_HEAP_STORE_WITH_CPHSTL_EXTRACT_H_
3
4   /*
5   Desc: The CPH STL extract(p) policy.
6   :     The method is optimized compared to original CHP STL
7   :     (and so is the extract of new weak queue too).
8   Auth: Asger Bruun 2009-2010.
9   */
10
11  #include "direct_heap_store.hpp"
12
13  namespace cphstl {
14
15    assertion_help(
16      static int count_yes = 0;
17      static int count_no = 0;
18    )
19
20    template <typename VCAEZ_CFG>
21    struct cph_stl_extract {
22      typedef typename VCAEZ_CFG::Z Z;
23      typedef typename Z::size_type size_type;
24      typedef typename Z::fast_number_type fast_number_type;
25      typedef typename Z::root_list_type root_list_type;
26      typedef typename VCAEZ_CFG::E E;
27      typedef typename E::hole_type hole_type;
28      typedef typename VCAEZ_CFG::H H;
29
30
31      static E* extract(H* hs) { return (*hs).decrement(); }
32
33      static void extract(H* hs, E* p) {
34        // Desc: CPH STL style priority queue extract(p).
35        // Opti: extra stack replaced with 'construct'.
36        assert(careful); // works only when careful
37        assert((*hs).is_valid());
38
39        E* replacement(extract(hs));
40        if (p == replacement) { return; }
41
42        assertion_help(bool was_root((*p).is_root());)
43        hole_type hole(p,true); // (auto splices out when needed)
44        assert(hole.is_root() == was_root);
45
46        // ?: simple debug mesurement of opti if(!hole.is_leaf) or if replacement compares
47        assertion_help(
48          //if((*p).is_leaf()) // --> 30% yes
49          //if((*hs).get_comparator()((*p).element(), (*replacement).element())
50          //  && !(*p).is_leaf()) // unescesarry construct --> 7%
51          if((*hs).get_comparator()((*p).element(), (*replacement).element())) // unescesarry
                construct --> 21%
52          {
53            ++count_yes; // __break_here;
54          } else {
55            ++count_no;
56          }
57        )
58
59        size_type n;
60        E* r((*p).construct(n, constant<bool,false>()));
61        root_list_type l(r);
```

```
62          while(!l.empty()) {
63            E* q(l.eject());
64            replacement = (*replacement).join(q, (*hs).get_comparator());
65          }
66
67          if(hole.do_splice()) {
68            (*replacement).splice_in(hole);
69            if(!hole.is_root()) (*hs).after_increase(replacement, (E*) hole.da);
70          } else {
71            (*hs).replace(p, replacement);
72          }
73
74          assert(!careful || (*p).is_root());
75          assert((*p).height() == 0);
76          assert((*hs).is_valid());
77        }
78      };
79    }
80    #endif
```

A.16 heap_store_with_fast_top.hpp

```
1   #ifndef _CPHSTL_HEAP_STORE_WITH_FAST_TOP_H_
2   #define _CPHSTL_HEAP_STORE_WITH_FAST_TOP_H_
3
4   /*
5   Desc: Wraps a heap store with cached find_top().
6   :     Refactored out of direct_heap_store.
7   Auth: Asger Bruun 2009-2010.
8   Note: This has to be the outer most wrapper,
9   :     in case the heap_store is wrapped in
10  :     other ways, eg. cormen_extract.
11  */
12
13  #include "direct_heap_store.hpp"
14
15  namespace cphstl {
16
17    template <typename B = direct_heap_store<> >
18    class with_fast_top : public B {
19      //typedef B B;
20      typedef with_fast_top<B> this_type;
21      typedef typename B::encapsulator_type E;
22      typedef typename E::value_type V;
23      typedef typename B::comparator_type C;
24      typedef typename B::allocator_type A;
25      //typedef typename B::size_type size_type;
26
27      E* top_;
28
29    public:
30
31      typedef typename B::size_type size_type;
32
33      template<typename U>
34      struct replace_comparator {
35        typedef with_fast_top<typename B::template replace_comparator<U>::other> other;
36      };
37
38      template<template <typename T> class U>
39      struct replace_comparator_template {
40        typedef with_fast_top<typename B::template replace_comparator_template<U>::other>
              other;
41      };
42
43      template<template <typename T> class U>
44      struct wrap_comparator {
45        typedef with_fast_top<typename B::template wrap_comparator<U>::other> other;
46      };
47
48      template<template <typename T> class U>
49      struct wrap_allocator {
50        typedef with_fast_top<typename B::template wrap_allocator<U>::other> other;
```

```
51      };
52
53      template<typename A5>
54      struct replace_allocator {
55        typedef with_fast_top<typename B::template replace_allocator<A5>::other> other;
56      };
57
58      // structors
59
60      with_fast_top()
61        : B(), top_(0) { // Meldable-priority-queue interface
62      }
63
64      // properties
65
66      size_type footprint(size_type n) const {
67        return sizeof(this_type);
68      }
69
70      E* top() const { return top_; }
71
72      // modifiers
73
74      void insert(E* p) {
75        B::insert(p);
76        if (top_ == 0 || B::get_comparator()((*top_).element(), (*p).element())) {
77          top_ = p;
78        }
79      }
80
81      E* extract() {
82        assert((*this).is_valid());
83        E* p(B::extract());
84        assert((*this).is_valid());
85        if(p == top_) { // do this to save compares.
86          if(B::empty()) top_ = 0;
87          else p = B::swap_top(p);
88          assert((*this).is_valid());
89        }
90        // if(p == top_) top_ = B::find_top(); // do this if no swap top.
91        assert((*this).is_valid());
92        return p;
93      }
94
95      void extract(E* x) {
96        // !: Use assertion bellow to test for stable sort (top  ==> is_root).
97        //assert(!((x == top_) && !((x == top_) && (*x).is_root()))); // is_stable
98
99        if(x == top_) top_ = B::extract_top(x);
100       else B::extract(x);
101       // or:
102       //   B::extract(x);
103       //   if(x == top_) top_ = B::find_top();
104     }
105
106     void increase(E* p, V const& v) {
107       B::increase(p, v);
108       if (B::get_comparator()((*top_).element(), (*p).element())) {
109         top_ = p;
110       }
111     }
112
113     template<typename PA // protected access
114     > void pop(PA& pa) {
115       E* p(top_);
116       top_ = B::extract_top(top_);
117       pa.destroy(p);
118     }
119
120     void meld(this_type& h2) {
121       B::meld(h2);
122       if (top_ != 0 && h2.top_ != 0
123         && B::get_comparator()((*top_).element(), (*h2.top_).element())) {
124           top_ = h2.top_;
```

```
125         }
126         h2.top_ = 0;
127       }
128
129       void swap(this_type& h2) {
130         B::swap(h2);
131         std::swap(top_, h2.top_);
132       }
133       void show() {
134         std::cout << (*this);
135       }
136
137     private:
138
139       friend std::ostream & operator <<
140         (std::ostream & os, this_type const& hs)
141       {
142         if(hs.top_ == 0) os << "top(nil)";
143         else os << "top(" << (*hs.top_).element() << ")";
144         os << "␣" << static_cast<B const&>(hs);
145         return os;
146       }
147     };
148   }
149   #endif
```

A.17 heap_store_with_vuillemin_extract.hpp

```
1   #ifndef _CPHSTL_HEAP_STORE_WITH_VUILLEMIN_EXTRACT_H_
2   #define _CPHSTL_HEAP_STORE_WITH_VUILLEMIN_EXTRACT_H_
3
4   /*
5   Desc: The original Vuillemin extract(p) policy.
6   Auth: Asger Bruun 2009-2010.
7   */
8
9   #include "direct_heap_store.hpp"
10
11  namespace cphstl {
12
13    template <typename VCAEZ_CFG>
14    struct vuillemin_extract {
15      typedef typename VCAEZ_CFG::Z Z;
16      typedef typename Z::size_type size_type;
17      typedef typename Z::fast_number_type fast_number_type;
18      typedef typename Z::root_list_type root_list_type;
19      typedef typename VCAEZ_CFG::E E;
20      typedef typename VCAEZ_CFG::H H;
21
22
23      static E* isolate_and_remove(H* hs, E* p, size_type& n) {
24        // Desc: Helper for Vuillemin extract(p). Writes tree size in n.
25        E* d((*p).distinguished_ancestor());
26        if(d == 0) {
27          n = (*hs).remove_root(p);
28          return 0;
29        } else {
30          E* r(isolate_and_remove(hs, d, n));
31          while(p != (*d).most_significant_child()) {
32            E* s((*d).split());
33            r = fast_number_type::inject(r,s);
34          }
35          (*d).split(); // detach p from d
36          r = fast_number_type::inject(r,d);
37          return r;
38        }
39      }
40
41      static void extract(H* hs, E* p) {
42        // Desc: the original Vuillemin extract(p).
43        // Opti: faster root (and top) extraction.
44        assert((*hs).is_valid()); assert((*p).is_valid());
45
```

```
46        if((*p).is_root()) {
47          size_type n((*hs).remove_root(p));
48          assert((*hs).is_valid());
49          if(!(*p).is_leaf()) {
50            size_type dummy;
51            E* r((*p).construct(dummy, constant<bool,false>()));
52            fast_number_type hs2(r, n-1);
53            (*hs).add(hs2);
54          }
55        } else  {
56          size_type n;
57          E* s(isolate_and_remove(hs, p, n)); // Hn-1, Hn-2,..Hp.
58
59          assert((*p).is_root());
60          E* t((*p).most_significant_child());
61          if(t != 0) {
62            size_type dummy;
63            E* r((*p).construct(dummy, constant<bool,false>())); // Hp-1,..H0.
64            s = fast_number_type::inject_range(s,r,t);
65          }
66          fast_number_type hs2(s, n-1);
67          (*hs).add(hs2);
68        }
69        assert((*p).is_root());
70        assert((*p).height() == 0);
71        assert((*hs).is_valid());
72      }
73    };
74  }
75  #endif
```

A.18 join_schedule_policies.hpp

```
1   #ifndef __CPHSTL_JOIN_SCHEDULE_POLICIES_H__
2   #define __CPHSTL_JOIN_SCHEDULE_POLICIES_H__
3   /*
4   Desc: Alternative join schedule policies for the redundant binary number.
5   :     (refactored out of redundant_binary_number_system.hpp)
6   :     They are named "fat", "medium", "light" and "ultra light".
7   :     None of the policies except from medium use dynamic memory
8   :     allocation, once instantiated. The medium weight policy is
9   :     the standard CPH STL dynamic memory allocating join schedule.
10  :     The light and ultralight relies on bitscanning rootlists.
11  :     The fat policy uses a stack based on a preallocated array.
12  Auth: Asger Bruun 2009-2010
13
14  Ref:  Amr Elmasry, Claus Jensen, and Jyrki Katajainen.
15  :     Relaxed weak queues: An alternative to run-relaxed heaps.
16  :     CPH STL Report 2005-2, Department of Computer Science,
17  :     University of Copenhagen (2005).
18  */
19
20  #include <deque>
21  #include <stack> // for the medium policy
22  //#include <vector> // for the fat policy
23  #include <array> // for the fat policy
24  #include <climits> // CHAR_BIT for the fat policy
25
26
27  #include "redundant_binary_number_system.hpp"
28
29
30  namespace cphstl {
31
32    template <int N> struct static_logb {
33      enum { value = (static_logb<N/2>::value + 1) };
34    };
35    template <> struct static_logb<1> {
36      enum { value = 0 };
37    };
38
39    template<typename E>
40    class join_schedule_ultralight {
```

```
41      // Desc: ultralight does all its work in propagate_carry.
42      //       (no real stack exists in this version)
43      // Req.: correct accu and carry before propagate_carry only.
44    public:
45
46      typedef typename E::height_type bit_pos_type;
47      typedef typename E::pointer pointer;
48
49      join_schedule_ultralight() {}
50
51      pointer top_node() { return 0; }
52      void clear() {}
53      void swap(join_schedule_ultralight & other) {
54      }
55
56      // Add a least significant carry bit.
57      void push(bit_pos_type, pointer) {}
58      template <typename L> bool is_valid(L*) { return true; }
59
60      // Remove least significant carry bit.
61      template <typename L> void pop(L*) {}
62
63      // Remove one specific carry bit.
64      template <typename L> void drop(L*, bit_pos_type) {}
65
66      // Scan and remove cancelled carry bits.
67      template <typename L> void scan_drop(L*) {}
68
69      // Repairs schedule after one increase to top.
70      void replace(pointer, pointer) {}
71
72      template <typename L>
73      void propagate_carry(L* t) {
74        /* Desc: Propagate the smallest carry bit if any.
75        Psedu:
76        1. locate accu and carry in root list and remember bit position e;
77        2. remove the weakest node from root list and join with the other;
78        3. propagate the carry bit at position e;
79        */
80
81        assert((*t).is_valid());
82        if((*t).size_.carry) {
83          pointer p((*t).begin()), z((*t).end());
84
85          typename L::bit_scanner bs((*t).size_, 0);
86          while(!bs.is_carry()) {
87            bs.next_bit(); z = p; p = L::next_get(p);
88          }
89
90          pointer q = L::next_get(p);
91          if (!L::comparator((*p).element(), (*q).element())) {
92            (*t).remove_after(p);
93            (*p).join(q);
94            if(!(*q).prev_rootable()) // a first single root
95              (*t).head_set(p);
96          } else {
97            (*t).remove_after(z);
98            (*q).join(p);
99            if(!(*p).prev_rootable())
100             (*t).head_set(q);
101         }
102         (*t).size_.propagate(bs.current());
103         assert((*t).is_valid());
104       }
105     }
106   };
107
108   ////////////////////////////////////////////////
109
110   template<typename E>
111   class join_schedule_light {
112     // Desc: this schedule has a better remove_root (eg. for vuillemin extract).
113     // Req.: correct accu and carry before every operation.
114   public:
```

```
115
116       typedef typename E::height_type bit_pos_type;
117       typedef typename E::size_type size_type;
118       typedef typename E::pointer pointer;
119
120       static const bit_pos_type none = ~ bit_pos_type(0);
121
122     private:
123
124       pointer node_;
125       bit_pos_type pos_;
126
127     public:
128
129       join_schedule_light() : node_(0), pos_(none) {}
130
131       pointer top_node() { return node_; }
132       bit_pos_type top_pos() { return pos_; }
133
134       void clear() { node_ = 0; pos_ = none; }
135       void swap(join_schedule_light & other) {
136         std::swap((*this).node_, other.node_);
137         std::swap((*this).pos_, other.pos_);
138       }
139
140       void push(bit_pos_type carry_pos, pointer p) {
141         assert(carry_pos >= 0);
142         assert(pos_ == none || carry_pos < pos_);
143         node_ = p; pos_ = carry_pos;
144       }
145
146       template <typename L> bool is_valid(L* t) {
147         if (pos_ == none) return ((*t).size_.carry == 0);
148         assert((*t).size_.carry != 0);
149
150         typename L::bit_scanner bs((*t).size_,0);
151         pointer p((*t).begin());
152         while(!bs.is_carry()) {
153           bs.next_bit(); p = L::next_get(p);
154         }
155         assert(p == node_);
156         return (p == node_);
157       }
158
159       template <typename L>
160       void pop(L* t) {
161         // assert(carry bits have been calculated).
162         assert(pos_ >= 0); assert(node_ != 0); assert(pos_ != none);
163         node_ = L::next_get(node_);
164         typename L::bit_scanner bs((*t).size_, (*t).js.pos_);
165         if(bs.has_more_carry()) {
166           while(!bs.is_carry()) {
167             bs.next_bit(); node_ = L::next_get(node_);
168           }
169           pos_ = bs.bit_pos();
170         } else { node_ = 0; pos_ = none; }
171       }
172
173       template <typename L>
174       void drop(L* t, bit_pos_type carry_pos) {
175         assert(carry_pos >= 0);
176         if(carry_pos == pos_) pop(t);
177       }
178
179       template <typename L>
180       void scan_drop(L* t) {
181         while((pos_ != none) && !bit_test((*t).size_.carry, pos_))
182           pop(t);
183       }
184
185       void replace(pointer replacement, pointer d) {
186         assert(replacement != 0); assert(d != 0);
187         if(node_ == d) node_ = replacement;
188       }
```

```
189
190       template <typename L>
191       void propagate_carry(L* t) {
192         assert((*t).is_valid());
193         if((*t).size_.carry) {
194           pointer p(node_);
195           pointer q = L::next_get(p);
196           bit_pos_type carry_pos(pos_);
197           (*t).size_.propagate(1 << pos_);
198           (*t).js.pop(t);
199           if (!L::comparator((*p).element(), (*q).element())) {
200             (*t).remove(q); // root_list::remove
201             (*p).join(q);
202           } else {
203             (*t).remove(p);
204             p = (*q).join(p);
205           }
206           if(bit_test((*t).size_.carry, carry_pos+1))
207             (*t).js.push(carry_pos+1, p);
208           assert((*t).is_valid());
209         }
210       }
211     };
212
213     ////////////////////////////////////////////////
214
215     template<typename E>
216     class join_schedule_medium {
217       // Desc: this schedule is the one from the cph stl weak queue.
218     public:
219
220       typedef typename E::size_type size_type;
221       typedef typename E::height_type bit_pos_type;
222       typedef typename E::pointer pointer;
223
224       static const bit_pos_type none = ~ bit_pos_type(0);
225
226     private:
227
228       struct stack_entry {
229         pointer carry_node;
230         bit_pos_type carry_pos;
231
232         stack_entry(pointer carry_node, bit_pos_type carry_pos)
233           : carry_node(carry_node), carry_pos(carry_pos) {}
234       };
235
236       typedef typename E::allocator_type::template
237         rebind<stack_entry>::other allocator_type;
238
239       typedef std::stack<stack_entry
240         , std::deque<stack_entry, allocator_type> > stack_type;
241
242       stack_type stk_;
243
244     public:
245
246       join_schedule_medium() : stk_() {}
247
248       pointer top_node() {
249         return stk_.empty()?0:stk_.top().carry_node;
250       }
251       bit_pos_type top_pos() {
252         return stk_.empty()?none:stk_.top().carry_pos;
253       }
254
255       void clear() { while(!stk_.empty()) stk_.pop(); }
256       void swap(join_schedule_medium & other) {
257         std::swap((*this).stk_, other.stk_);
258       }
259
260       void push(bit_pos_type carry_pos, pointer p) {
261         assert(carry_pos >= bit_pos_type(0) && carry_pos <= bit_pos_type(sizeof(size_type)*8));
262         if(!stk_.empty()) { assert(carry_pos < stk_.top().carry_pos); }
```

```
263         stack_entry e(p,carry_pos);
264         stk_.push(e);
265       }
266
267       template <typename L> bool is_valid(L*) { return true; }
268
269       template <typename L>
270       void pop(L* /*ignored*/) {
271         assert(!stk_.empty());
272         stk_.pop();
273       }
274
275       template <typename L>
276       void drop(L* /*ignored*/, bit_pos_type carry_pos) {
277         stack_type tmp;
278         while(!stk_.empty()) {
279           if(stk_.top().carry_pos == carry_pos) {
280             stk_.pop();
281             break;
282           } else {
283             tmp.push(stk_.top());
284             stk_.pop();
285           }
286         }
287         while(!tmp.empty()) {
288           stk_.push(tmp.top());
289           tmp.pop();
290         }
291       }
292
293       template <typename L>
294       void scan_drop(L* t) {
295         stack_type tmp;
296         while(!stk_.empty()) {
297           if(!bit_test((*t).size_.carry, stk_.top().carry_pos))
298             stk_.pop();
299           else {
300             tmp.push(stk_.top());
301             stk_.pop();
302           }
303         }
304         while(!tmp.empty()) {
305           stk_.push(tmp.top());
306           tmp.pop();
307         }
308       }
309
310       void replace(pointer replacement, pointer d) {
311         stack_type tmp;
312         while(!stk_.empty()) {
313           if(stk_.top().carry_node == d) {
314             stack_entry e(replacement, stk_.top().carry_pos);
315             tmp.push(e);
316             break;
317           } else {
318             tmp.push(stk_.top());
319             stk_.pop();
320           }
321         }
322         while(!tmp.empty()) {
323           stk_.push(tmp.top());
324           tmp.pop();
325         }
326       }
327
328       template <typename L>
329       void propagate_carry(L* t) {
330         assert((*t).is_valid());
331         if((*t).size_.carry) {
332           pointer p((*t).js.top_node());
333           pointer q(L::next_get(p));
334           if (!L::comparator((*p).element(), (*q).element())) {
335             (*t).remove(q); // root_list::remove
336             (*p).join(q);
```

```
337           } else {
338             (*t).remove(p);
339             p = (*q).join(p);
340           }
341           bit_pos_type curr_pos((*t).js.top_pos());
342           (*t).js.pop(t);
343           (*t).size_.propagate(size_type(1) << curr_pos);
344           ++curr_pos;
345           if(bit_test((*t).size_.carry, curr_pos))
346             (*t).js.push(curr_pos, p);
347           assert((*t).is_valid());
348         }
349       }
350     };
351
352     ////////////////////////////////////////////////
353
354     template<typename E>
355     class join_schedule_fat {
356       // Desc: this schedule is optimised for cph stl extract.
357     public:
358
359       typedef typename E::size_type size_type;
360       //typedef typename E::height_type bit_pos_type; // GCC doesn't like.
361       typedef unsigned int bit_pos_type;
362       typedef typename E::pointer pointer;
363
364       static const bit_pos_type none = ~ bit_pos_type(0);
365
366     private:
367
368       struct stack_entry {
369         pointer carry_node;
370         bit_pos_type carry_pos;
371       };
372
373       static const bit_pos_type  max_stack
374         = (sizeof(size_type)*CHAR_BIT/2) - (static_logb<sizeof(E)>::value/2);
375
376       // std::stack is not an option because we need random access.
377       std::array<stack_entry,max_stack> stk_;
378       bit_pos_type top_;
379       // is better than: stack_entry stk_[max_stack];
380
381       size_type bit_val(bit_pos_type pos) {
382         assert(pos != none);
383         return size_type(1) << pos;
384       }
385
386     public:
387
388       join_schedule_fat() : /*stk_(max_stack),*/ top_(none) {}
389
390       pointer top_node() {
391         return top_ == none?0:stk_[top_].carry_node;
392       }
393       bit_pos_type top_pos() {
394         return top_ == none?none:stk_[top_].carry_pos;
395       }
396
397       void clear() { top_ = none; }
398
399       void swap(join_schedule_fat & other) {
400         std::swap((*this).top_, other.top_);
401         (*this).stk_.swap(other.stk_);
402       }
403
404       void push(bit_pos_type carry_pos, pointer p) {
405         assert(carry_pos >= 0 && carry_pos <= (sizeof(size_type)*8));
406         if(top_ != none) {
407           assert(carry_pos < stk_[top_].carry_pos);
408           ++top_;
409         } else top_ = 0;
410         stk_[top_].carry_pos = carry_pos;
```

```
411         stk_[top_].carry_node = p;
412       }
413
414       template <typename L> bool is_valid(L*) { return true; }
415
416       template <typename L>
417       void pop(L* /*ignored*/) {
418         assert(top_ != none);
419         if(top_ == 0) top_ = none;
420         else --top_;
421       }
422
423       template <typename L>
424       void drop(L* /*ignored*/, bit_pos_type carry_pos) {
425         if(top_ != none) {
426           bit_pos_type i(0);
427           while(stk_[i].carry_pos != carry_pos && i != (top_+1)) {
428             assert(i <= top_);
429             ++i;
430           }
431           if(i <= top_) {
432             while(i != top_) {
433               stk_[i] = stk_[i+1];
434               ++i;
435             }
436             if(top_ == 0) top_ = none;
437             else --top_;
438           }
439         }
440       }
441
442       template <typename L>
443       void scan_drop(L* t) {
444         assert(top_ != none);
445         bit_pos_type i(0);
446         for(bit_pos_type j(1), e(top_+1); j < e; ++j) {
447           if(((bit_val(stk_[i].carry_pos)) & (*t).size_.carry) != 0)
448             ++i;
449           else if(((bit_val(stk_[j].carry_pos)) & (*t).size_.carry) != 0)
450             stk_[i] = stk_[j];
451         }
452         if(((bit_val(stk_[i].carry_pos)) & (*t).size_.carry) != 0) top_ = i;
453         else if(i == 0) top_ = none;
454         else top_ = i-1;
455       }
456
457       void replace(pointer replacement, pointer d) {
458         if(top_ != none) {
459           for(bit_pos_type i(0); i <= top_; ++i) {
460             if(stk_[i].carry_node == d) {
461               stk_[i].carry_node = replacement;
462               break;
463             }
464           }
465         }
466       }
467
468       template <typename L>
469       void propagate_carry(L* t) {
470         assert((*t).is_valid());
471         if((*t).size_.carry) {
472           assert(top_ != none);
473           pointer p((*t).js.top_node());
474           pointer q(L::next_get(p));
475           if (!L::comparator((*p).element(), (*q).element())) {
476             (*t).remove(q); // (ie. root_list::remove)
477             (*p).join(q);
478             if(!(*q).prev_rootable()) // is_leaf
479               (*t).head_set(p);
480           } else {
481             (*t).remove(p);
482             (*q).join(p);
483             if(!(*p).prev_rootable()) // is_leaf
484               (*t).head_set(q);
```

```
485             p = q;
486           }
487           bit_pos_type curr_pos((*t).js.top_pos());
488           (*t).js.pop(t);
489           (*t).size_.propagate(bit_val(curr_pos));
490           ++curr_pos;
491           if(((*t).size_.carry & (bit_val(curr_pos))) != 0)
492             (*t).js.push(curr_pos, p);
493         }
494         assert((*t).is_valid());
495       }
496     };
497   }
498   #endif
```

A.19 light_binomial_node.hpp

```
1   /*
2   Desc: A binomial two-pointer non templated node base.
3   Auth: Jyrki Katajainen, Asger Bruun © 2009
4   Stat: 20091025 is_root isnt safe any more!
5   Note: Owner-pointer are offered only from pseudo root.
6         This structure uses one out of several possible
7         2-pointer arrangements, where only the unidirectional
8         root list version are fully defined.
9   Ref : We just drew the node structure on the black board.
10        It offers a nice successor function in const time.
11        The design is obvious, but we find no references.
12        20091029 We exercised this structure one more time
13        on the black board and realised that the path from
14        leaf to root is lg(n)^2, ie. probably of no
15        practical use, though perhaps beautiful.
16
17  *   o
18  *   !
19  *   o-----
20  *   :     \
21  *   o-o-o---o--
22  *     | :\   :  \
23  *     o  o-o  o-o-o
24  *          |    | :\
25  *          o    o  o-o
26  *                  |
27  *                  o
28  */
29
30
31  #ifndef _CPHSTL_PQFW_BINOMIAL_TWO_PTR_NODE_H_
32  #define _CPHSTL_PQFW_BINOMIAL_TWO_PTR_NODE_H_
33
34  #include "assert.h++" // assert
35  #include <cstddef> // std::size_t
36  #include <string> // std::string
37  #include <iostream> // std::ostream + std::ostringstream
38  #include <sstream> // std::sstream
39
40  #include "root_list.hpp"
41
42  namespace cphstl {
43    namespace pqfw_node {
44      class light_binomial_node_base {
45        typedef light_binomial_node_base N;
46
47        union {
48          void* owner_;
49          N* s_; // sibling or parent
50          // s ~ child?pwh-left:pwh-distinguished-ancestor
51        };
52        N* child_; // ~ pwh-right
53
54        light_binomial_node_base(N const&);
55        N& operator=(N const&);
56
```

```
57      public:
58
59        typedef light_binomial_node_base node_base;
60        typedef std::size_t size_type;
61        typedef unsigned char height_type; // assert(size_of(void*) <= (256/8));
62        //typedef root_list<N> root_list_type;
63
64        struct hole_type {
65          enum placements {empty, root, right, wrong};
66          placements placement;
67          N* da;
68          union {
69            N* q; // bino parent or prev_sibling / weak right or wrong parent
70            void* owner;
71          };
72
73          hole_type() : placement(empty), da(0), q(0) {}
74
75          hole_type(N* n, bool auto_splice_out, bool forced=false)
76            : da((*n).distinguished_ancestor())
77          {
78            assert(n != 0);
79            if(da==0) {
80              placement = root;
81              owner = (*n).owner_;
82            } else {
83              N* p = (*n).parent(); assert(p != 0);
84              N* ps = (*n).prev_sibling(p);
85              placement = (ps == 0)?right:wrong;
86              q = (ps == 0)?p:ps;
87            }
88            assert(is_valid());
89            if((auto_splice_out && do_splice()) || forced) {
90              assert(is_valid());
91              switch(placement) {
92                case hole_type::root: (*n).owner_ = 0; break;
93                case hole_type::right: (*q).right_split(); break;
94                case hole_type::wrong: (*q).wrong_split(); break;
95                default: assert(0);
96              }
97              assert(is_root());
98            }
99          }
100         bool is_root() const { return da==0; }
101         bool do_splice() { return !is_root(); }
102         bool is_valid() const {
103           bool hole_is_valid = this!=0 && placement!=empty
104             && ((placement!=root && q!=0 && (*q).is_valid())
105             || placement==root);
106           assert(hole_is_valid);
107           return hole_is_valid;
108         }
109       };
110       hole_type splice_out(bool forced=false) {
111         hole_type hole(this,true,forced);
112         return hole;
113       }
114       template <typename H>
115       void splice_in(hole_type& hole, H& /*heap_store*/,bool forced=false) {
116         if(!(hole.do_splice() || forced)) return;
117         assert(!careful || is_root());
118         assert(hole.is_valid());
119         switch(hole.placement) {
120           case hole_type::root:
121             (*this).owner_ = hole.owner;
122             break;
123           case hole_type::right: (*hole.q).right_join(this); break;
124           case hole_type::wrong: (*hole.q).wrong_join(this); break;
125           default: assert(0);
126         }
127         hole.placement = hole_type::empty;
128       }
129       void splice_in(hole_type& hole) {
130         if(!hole.do_splice()) return;
```

```
131           assert(!careful || is_root());
132           assert(hole.is_valid());
133           switch(hole.placement) {
134             case hole_type::root:
135               (*this).owner_ = hole.owner;
136               break;
137             case hole_type::right: (*hole.q).right_join(this); break;
138             case hole_type::wrong: (*hole.q).wrong_join(this); break;
139             default: assert(0);
140           }
141           hole.placement = hole_type::empty;
142         }
143
144
145         light_binomial_node_base() : s_(0), child_(0) {}
146
147
148         // properties.
149
150         static size_type footprint() { return sizeof(N); }
151         bool is_valid() const { return this!=0; }
152         //bool has_parent() const { return parent()!=0; }
153         bool is_other_root() const {
154           assert(is_valid());
155           return s_ != 0 && ((*s_).s_ == this);
156         }
157         bool has_child(N const* p) const {
158           //?? assert(child_ != 0);
159           assert(is_valid());
160           assert((*p).is_valid());
161           bool yes(false);
162           for(N* c(child_); c != 0 && c != this; c = (*c).s_) {
163             if(p == c) { yes = true; break; }
164           }
165           return yes;
166         }
167         bool is_root() const {
168           assert(is_valid());
169           if (s_ == 0) return true; // clean and isolated root.
170           if((*s_).s_ == this)  return true; // root with pseudo root or pseudo root.
171           if(child_ == 0) return !(*s_).has_child(this); // root listed root with zero height
                .
172
173           N* p = (N*) this; // locate parent if any
174           while (p != 0) {
175             if((*p).child_ == 0) break;
176             p = (*p).s_;
177           }
178           if(p == 0) return true; // root listed root heigher than zero.
179           assert((*p).s_ != 0);
180           return (!(*(*p).s_).has_child(this));
181         }
182
183         bool is_leaf() const { assert(is_valid()); return child_ == 0; }
184         bool is_marked() const { assert(is_valid()); return false; } // ??
185
186         void* owner() const { assert((*this).is_root()); return owner_; }
187         void*& owner() { assert((*this).is_root()); return owner_; }
188
189         N* root() const {
190           // Desc: find root. T(n): O(lg(n)). ES: no throw.
191           assert(is_valid());
192           N* p = (N*) this;
193           while (! (*p).is_root()) { p = (*p).s_; }
194           return p;
195         }
196
197         N* distinguished_ancestor() const {
198           assert(is_valid());
199             return parent();
200         }
201         N* distinguished_descendant() const {
202           assert(is_valid());
203           return (*this).is_root()?child():sibling();
```

```
204         }
205
206         N const* successor() const {
207           // Desc: "child first"-ordered traversal.
208           // T(n): O(1). ES: no throw.
209           assert(is_valid());
210           if(child_ != 0) return child_;
211           else if (s_ != 0)
212             return (*s_).s_;
213           else return 0;
214         }
215         N * successor() {
216           // Desc: "child first"-ordered traversal.
217           // T(n): O(1). ES: no throw.
218           assert(is_valid());
219           if(child_ != 0) return child_;
220           else if (s_ != 0)
221             return (*s_).s_;
222           else return 0;
223         }
224
225         height_type height() const {
226           // Desc: Slow height, usefull for assertions. ES: no throw.
227           assert(is_valid());
228           height_type h = 0;
229           for(N* c = child_; c != 0; c = (*c).child_) { ++h; }
230           return h;
231         }
232
233         bool tree_valid() const {
234           // Desc: check sub-structure (when debugging).
235           if(this == 0)
236             return false;
237
238           if(child() != 0) {
239             N* p = (*child()).parent();
240             if(p != this)
241               return false;
242             if(!(*child()).tree_valid())
243               return false;
244             if(sibling() == 0)
245               return true;
246             else
247               if(!(*sibling()).tree_valid())
248                 return false;
249
250             return true;
251           } else return true;
252         }
253
254         // operations.
255
256         N* join(N* weaker) {
257           // Desc: join with a weaker tree. T(n): O(1). ES: no throw.
258           // Note: the owner should ensure "assert((*this).height() == (*weaker).height())";
259           assert(!careful || is_root());
260           right_join(weaker);
261           return this;
262         }
263
264         N* most_significant_child() {
265           // Desc: return the potentiel return of a split.
266           assert(is_valid());
267           return (*this).child_;
268         }
269         N* split() {
270           // Desc: split weaker tree. T(n): O(1). ES: no throw.
271           assert(is_root());
272           return right_split();
273         }
274
275         template <class Boolean>
276         N* construct(size_type& n, const Boolean calc_size) {
277           // Desc: Vuillemin "construct". T(n): O(lg n). ES: no throw.
```

```
278           //        Isolates the root from its childs.
279           // Ret:   A root list in reverse order of an recusive split
280           //        + total number of nodes including the root.
281           assert(!careful || is_root());
282           N* r(0);
283           if(calc_size) n = 1;
284           for(N* c(child_); c != 0 && c != this; ) {
285             N* d((*c).s_);
286             (*c).s_ = r;
287             r = c;
288             if(calc_size) n <<= 1;
289             c = d;
290           }
291           child_ = 0;
292           return r;
293         }
294
295         N* swap(N* p) {
296           // general swap not (yet) supported.
297           assert(p == (*this).distinguished_ancestor());
298           return promote(p);
299         }
300
301         void swap_roots(N* q) {
302           assert(is_root());
303           assert((*q).is_root());
304
305           N* p = this;
306           std::swap((*p).child_, (*q).child_);
307           N* c = (*q).child_;
308           while (c != 0 && (*c).s_ != p) { c = (*c).s_; }
309           if(c != 0) (*c).s_ = q;
310           c = (*p).child_;
311           while (c != 0 && (*c).s_ != q) { c = (*c).s_; }
312           if(c != 0) (*c).s_ = p;
313         }
314
315         N* promote_alt(N* p) {
316           // Vuillemin compatible.
317           promote(p);
318           return (*this).distinguished_ancestor();
319         }
320
321         N* promote(N* p) {
322           // CPH STL compatible.
323           /*
324           * Desc: exchange position with distinguished_ancestor.
325           * T(n): O(lg(n)). ES: no throw.
326           * Note: binomial parent = distinguished_ancestor.
327           *
328           *         g
329           *         ^ \--------
330           *         |          \
331           *    (1)   <- <- p <- f   (2)
332           *                ^ \-----
333           *                |      \
334           *          (3)    <- n <- s    (4)
335           *                     ^ \
336           *                     |  \
337           *                      <-
338           *                          (5)
339           *
340           *    p: parent, s: oldest sibling,
341           *    g: grandparent, f: grand first child)
342           */
343           assert(is_valid() && parent()!=0);
344           assert(p == distinguished_ancestor()); // T=lg?
345
346           N* g = (*p).parent();
347           N* pc = (*p).child_;
348           N* n = this;
349           N* nc = (*n).child_;
350           N* nsp = (*n).s_;
351
```

```
352           // 1
353           (*n).s_ = (*p).s_;
354
355           // 2
356           if(g != 0) {
357             N* gc = (*g).child_;
358             if(p == gc) {
359               (*g).child_ = n;
360             } else {
361               while((*gc).s_ != p) gc = (*gc).s_;
362               (*gc).s_ = n;
363             }
364           }
365
366           // 3
367           if(nsp == p) {
368             (*p).s_ = n;
369           } else {
370             (*p).s_ = nsp;
371             while((*nsp).s_ != p) nsp = (*nsp).s_;
372             (*nsp).s_ = n;
373           }
374
375           // 4
376           if(n == pc) {
377             (*n).child_ = p;
378           } else {
379             (*n).child_ = pc;
380             while((*pc).s_ != n) pc = (*pc).s_;
381             (*pc).s_ = p;
382           }
383
384           // 5
385           (*p).child_ = nc;
386           if(nc != 0) {
387             while((*nc).s_ != n) nc = (*nc).s_;
388             (*nc).s_ = p;
389           }
390           return this;
391         }
392
393         std::string str() const { return (*this).str<N>(); }
394
395       protected:
396
397         template <typename D>
398         std::string str() const {
399           std::ostringstream os;
400           if(this == 0) os << "nil";
401           else {
402             N* x = (*this).child();
403             if(x != 0) {
404               os << "(" << (*static_cast<D *const>(x)).str();
405               for(x = (*x).s_; x != this; x = (*x).s_)
406                 os << "," << (*static_cast<D *const>(x)).str();
407               os << ")";
408             }
409           }
410           return os.str();
411         }
412
413         N* child() const { assert(is_valid()); return child_; }
414
415         N* sibling() const {
416           assert(is_valid());
417           return ((*this).child_ != 0)?s_:0;
418         }
419         N* parent() const {
420           // Desc: find parent. T(n): O(lg(n)). ES: no throw.
421           // Note: reliable when structure is valid binomial,
422           //       ie. dont use during a splice out.
423           assert(is_valid());
424           if(is_root()) return 0;
425           N* p = (N*) this;
```

```
426           while (p != 0 && (*p).child_ != 0) { p = (*p).s_; }
427           assert(p != 0); assert((*p).s_ != 0); // because !is_root
428           return (*p).s_;
429         }
430
431       private:
432
433         // subtree split and join.
434
435         N* right_join(N* weaker) {
436           // Desc: join with a weaker tree. T(n): O(1). ES: no throw.
437           assert(is_valid()); assert(this!=weaker);
438           assert(!careful || (*weaker).is_root());
439           if((*this).child_ != 0) (*weaker).s_ = (*this).child_;
440           else (*weaker).s_ = this;
441           (*this).child_ = weaker;
442           return this;
443         }
444
445         N* right_split() {
446           // Desc: split of weaker tree. T(n): O(1). ES: no throw.
447           assert(is_valid()); assert(child() != 0);
448           N* b = (*this).child_;
449           (*this).child_ = (*b).sibling();
450           (*b).s_ = 0;
451           return b;
452         }
453
454         // splice helpers.
455
456         N* wrong_join(N* younger) {
457           // Desc: insert sibling subtree. T(n): O(1). ES: no throw.
458           assert(is_valid() && (*younger).is_root() && this!=younger);
459           (*younger).s_=(*this).s_;
460           (*this).s_=younger;
461           return this;
462         }
463
464         N* wrong_split() {
465           // Desc: unlink next sibling subtree. T(n): O(1). ES: no throw.
466           assert(is_valid()); assert(sibling() != 0);
467           N* b = (*this).s_;
468           (*this).s_ = (*b).s_; (*b).s_ = 0;
469           return b;
470         }
471
472         N* prev_sibling(N* known_parent) const { // "light weak and binomial identical code"
473           // Desc: locate older sibling from parent or return 0.
474           //       (use prev_sibling(parent())
475           assert(is_valid()); assert(known_parent == (*this).parent());
476           if (known_parent == 0) return 0;
477           else {
478             N* c = (*known_parent).child_;
479             if(c == this) c = 0;
480             else while((*c).sibling() != this) c = (*c).sibling();
481             return c;
482           }
483         }
484       };
485
486       template <>
487       struct node_traits<light_binomial_node_base>  { enum { has_splice=1 }; enum { has_owner=0
              }; enum { is_root_listable=1 }; };
488   }
489 }
490
491 #endif
```

A.20 ms_c_fix.hpp

```
1  #ifndef __CPHSTL_MS_C_FIX_HPP__
2  #define __CPHSTL_MS_C_FIX_HPP__
3
```

```
4  /*
5  Desc: Enabling leak check and add missing ilog when MS Compiler.
6  Auth: Asger Bruun 2009-2010
7  */
8
9  #include <cstddef> // std::size_t
10
11 #ifdef _MSC_VER // msvc leak detection setup
12 #define NOMINMAX // use std::min & max.
13   #ifdef INCLUDE_LEAK_CHECK
14     #define _CRTDBG_MAP_ALLOC
15     #include <stdlib.h>
16     #include <crtdbg.h>
17     #include <iostream>
18     #include <typeinfo> // typeid
19
20     int checked_main(int argc, char** argv); // the user defined main
21     int main(int argc, char** argv) {
22       int i,j;
23       _CrtSetReportMode( _CRT_ASSERT, _CRTDBG_MODE_FILE);
24       // set your break points in your user defined main like this:
25       //   _CrtSetBreakAlloc(123);
26
27       j = checked_main(argc, argv);
28
29       // write leak report if leaks were detected.
30       _CrtSetReportMode( _CRT_WARN, _CRTDBG_MODE_FILE );
31       _CrtSetReportFile( _CRT_WARN, _CRTDBG_FILE_STDERR );
32       i = _CrtDumpMemoryLeaks();
33       if(i) {
34         std::cout << "leak␣detected.␣" << i << "\n";
35         system("pause");
36       }
37       return j;
38     }
39     #define main checked_main
40   #else
41     #include <stdlib.h>
42     #include <crtdbg.h>
43     #include <iostream>
44     #include <typeinfo> // typeid
45   #endif
46
47   #define system_pause() system("pause")
48
49 #else
50   #include <typeinfo> // typeid
51   #define _CrtSetBreakAlloc(ignored)
52   #define system_pause()
53 #endif
54
55 //#include "assert.h++"
56 #include "bit_manipulation.h++"
57
58 #ifdef _MSC_VER // add missing missing "ilogb", in msvc <cmath>.
59     //#include <intrin.h> // _BitScanForward
60     //#pragma intrinsic(_BitScanForward,_BitScanReverse,__popcnt)
61
62     #pragma warning(disable:4244)
63     #pragma warning(disable:4127)
64     #pragma warning(disable:4099)
65     //warning C4244: 'argument' : conversion from 'double' to 'size_t', possible loss of
           data
66     //warning C4127: conditional expression is constant
67     //warning C4099: 'std::pair' : type name first seen using 'struct' now seen using '
           class'
68
69     // the following warning indicates potential degraded debugging capabilities:
70     #pragma warning(disable:4503)
71     // warning C4503: decorated name length exceeded, name was truncated
72
73     // MSC has no ilogb:
74     static unsigned int my_ilogb(size_t x) { // integral reverse of pw2
75       // assert(x>= 1);
```

```
76        unsigned int ret = 0;
77        x = (x>>1); while(x != 0) { x = (x>>1); ++ret; }
78        return ret;
79      }
80      int ilogb(double x) throw() {
81        return my_ilogb(x);
82      }
83
84      namespace std {
85        //static unsigned int log2(size_t x) { return ilogb(x); }
86        unsigned int log2(size_t x) { return ilogb(x); }
87      }
88
89
90  #elif defined(__GNUC__)
91  #endif
92
93  #include <string>
94
95      std::string typeid_name_fix(std::type_info const& ti);
96
97      std::string str_replace(std::string s, std::string f, std::string r) {
98        for(size_t p = s.find(f); p != std::string::npos; p = s.find(f,p+r.length()))
99          s.replace(p, f.size(), r);
100       return s;
101     }
102
103 #if defined(__GNUC__)
104     #include <exception>
105     #include <iostream>
106     #include <cxxabi.h>
107
108     std::string typeid_name_fix(std::type_info const& ti) {
109       // demangling.
110       // ref: gcc.gnu.org/onlinedocs/libstdc++/manual/bk01pt12ch39.html
111       int      status;
112       char    *realname;
113       realname = abi::__cxa_demangle(ti.name(), 0, 0, &status);
114       // std::cout << ti.name() << "\t => " << realname << "\t: " << status << '\n';
115       std::string s;
116       if(status == 0) s = realname;
117       else s = ti.name();
118       free(realname);
119       return str_replace(str_replace(s
120         , "cphstl::pqfw_node::",""), "cphstl::","");
121     }
122 #else
123     std::string typeid_name_fix(std::type_info const& ti) {
124       std::string s(ti.name());
125       return str_replace(str_replace(str_replace(str_replace(str_replace(s
126         , "cphstl::benchmarking::","")
127         , "cphstl::pqfw_node::","")
128         , "cphstl::","")
129         , "class ","")
130         , "struct ","");
131     }
132 #endif
133
134 #endif // __CPHSTL_MS_C_FIX_HPP__
```

A.21 node_config.hpp

```
1  #ifndef _CPHSTL_PQFW_NODE_CONFIG_H_
2  #define _CPHSTL_PQFW_NODE_CONFIG_H_
3
4  /*
5  Desc: Node type conversion and iteration.
6  Auth: Asger Bruun 2009-2010
7  */
8
9  #include "assert.h++"
10 #include "ms_c_fix.hpp"
11
```

```
12 #include "node_with_direct_value.hpp"
13 #include "node_with_indirect_value.hpp"
14 #include "node_proxy.hpp"
15 #include "node_with_facade.hpp"
16
17 #include "pwh_node.hpp"
18 #include "binomial_node.hpp"
19 #include "light_binomial_node.hpp"
20 #include "brown_r_node.hpp"
21 #include "brown_k_node.hpp"
22
23 //#include "weak-heap-node.h++"
24 //#include "pennant-node.h++"
25 //#include "fat-weak-heap-node.h++"
26
27 namespace cphstl {
28   namespace pqfw_node {
29
30     template <typename V = char
31       , typename C = std::greater<V>
32       , typename A = std::allocator<V>
33     >
34     struct node_config {
35       template <typename B>
36       struct n {
37         typedef w_facade<w_direct_value<B,V,C,A> > dv;
38         typedef w_facade<w_indirect_value<B,V,C,A> > iv;
39         typedef w_facade<proxy<B,V,C,A> > px;
40       };
41
42       // The PQFW nodes are not any longer compatible:
43       // typedef weak_heap_node<V,A>      pqfw_std;
44       // typedef fat_weak_heap_node<V,A>  pqfw_fat;
45       // typedef pennant_node<V,A>        pqfw_pen;
46
47       typedef typename n<binomial_node_base>::dv bno_3_dv; // binomial tree
48       typedef typename n<pwh_node_base>::dv      pwh_3_dv; // perfect weak heap
49       typedef typename n<light_binomial_node_base>::dv  bno_2_dv;
50       typedef w_facade<brown_r_node<V,C,A> >            bno_r_dv;
51       typedef w_facade<brown_k_node<V,C,A> >            bno_k_dv;
52       typedef w_facade<brown_r_node<V,C,A,false> >       bno_r_dv_bidir;
53       typedef w_facade<brown_k_node<V,C,A,false> >       bno_k_dv_bidir;
54
55       typedef typename n<binomial_node_base>::iv        bno_3_iv;
56       typedef typename n<pwh_node_base>::iv             pwh_3_iv;
57       typedef typename n<light_binomial_node_base>::iv  bno_2_iv;
58       typedef w_facade<w_indirection<brown_r_node<V,C,A> > > bno_r_iv;
59       typedef w_facade<w_indirection<brown_k_node<V,C,A> > > bno_k_iv;
60
61       typedef typename n<binomial_node_base>::px        bno_3_px;
62       typedef typename n<pwh_node_base>::px             pwh_3_px;
63       typedef typename n<light_binomial_node_base>::px  bno_2_px;
64
65       template <template <typename E> class T>
66       static void execute() {
67         T<bno_3_dv>::execute("bno_3_dv");
68         T<pwh_3_dv>::execute("pwh_3_dv");
69         T<bno_2_dv>::execute("bno_2_dv");
70         T<bno_r_dv>::execute("bno_r_dv");
71         T<bno_k_dv>::execute("bno_k_dv");
72         T<bno_3_iv>::execute("bno_3_iv");
73         T<pwh_3_iv>::execute("pwh_3_iv");
74         T<bno_2_iv>::execute("bno_2_iv");
75         T<bno_r_iv>::execute("bno_r_iv");
76         T<bno_k_iv>::execute("bno_k_iv");
77         T<bno_3_px>::execute("bno_3_px");
78         T<pwh_3_px>::execute("pwh_3_px");
79         T<bno_2_px>::execute("bno_2_px");
80       }
81
82       /* to execute the above use a node-typed runner like:
83         template <typename E>
84         struct test {
85           static void execute(char const* friendly_type_name) { ...; }
```

```
86          };
87        */
88      };
89    }
90  }
91  #endif
```

A.22 node_proxy.hpp

```
1   #ifndef _CPHSTL_PQFW_NODE_PROXY_HPP_
2   #define _CPHSTL_PQFW_NODE_PROXY_HPP_
3
4   /*
5   Desc: Public interface is a proxy.
6   Feat: Exception safe element assignment.
7   Auth: Jyrki Katajainen, Asger Bruun © 2009
8   Stat: 2091010 tested ok with ms-compiler and gcc 4.3.3.
9   Prob: Discuss and find agreement on the allocator location.
10  *     -> (n)
11  *          |
12  *        [v]
13  */
14
15  #include <string> // std::string
16  #include <iostream> // std::ostream
17  #include <memory> // std::allocator
18
19  #include "light_binomial_node.hpp"
20  #include "node_with_direct_value.hpp"
21  #include "node_with_facade.hpp"
22
23  namespace cphstl {
24    namespace pqfw_node {
25      template < typename B
26        , typename V
27        , typename C
28        , typename A
29      >
30      class proxy;
31
32      template < typename B = light_binomial_node_base
33        , typename V = char
34        , typename C = std::less<V>
35        , typename A = std::allocator<V>
36      >
37      struct proxy_subject_element_type {
38        typedef proxy<B,V,C,A> P;
39        V value;
40        P* prx_ptr;
41        proxy_subject_element_type(V const& v, P* prx_ptr) : value(v), prx_ptr(prx_ptr) {}
42        proxy_subject_element_type(proxy_subject_element_type const& c) : value(c.value),
              prx_ptr(c.prx_ptr) {}
43      private:
44        proxy_subject_element_type& operator = (proxy_subject_element_type const& c);
45      };
46
47      template < typename B = light_binomial_node_base
48        , typename V = char
49        , typename C = std::less<V>
50        , typename A = std::allocator<V>
51      >
52      class proxy {
53        typedef proxy<B,V,C,A> PE;
54        typedef proxy_subject_element_type<B,V,C,A> subject_element_type;
55        typedef w_direct_value<B, subject_element_type> I;
56        I* subject_; // the real subject
57
58        proxy();
59        proxy(proxy const&);
60        proxy& operator = (proxy const&);
61
62      public:
63
```

```
64        static const bool promote_is_root_list_neutral = false;
65
66        typedef V value_type;
67        typedef C comparator_type;
68        typedef typename A::template rebind<I>::other allocator_type;
69        typedef typename I::size_type size_type;
70        typedef typename I::height_type height_type;
71        typedef typename I::hole_type hole_type;
72        typedef B node_base_type;
73        //typedef typename B::root_list_type root_list_type;
74
75      private:
76
77        static allocator_type allo_;
78
79      public:
80
81        proxy(V const& v) {
82          subject_ = allo_.allocate(1);
83          try { new(subject_) I(subject_element_type(v,this)); }
84          catch (...) { allo_.deallocate(subject_, 1); throw; }
85        }
86
87        ~proxy() {
88          (*subject_).~I();
89          allo_.deallocate(subject_, 1);
90        }
91
92        V const& element() const { return (*subject_).element().value; }
93
94        V& element() {
95          return (*subject_).element().value;
96        }
97
98        void element_safe_set(V const& v) {
99          I* i = allo_.allocate(1);
100         try {
101           new(i) I(subject_element_type(v,this)); // might throw.
102           (*subject_).swap(i);
103           std::swap(subject_,i);
104           allo_.deallocate(i, 1);
105         }
106         catch (...) { allo_.deallocate(i, 1); throw; }
107       }
108
109       // properties.
110       static typename B::size_type footprint() {
111         return sizeof(PE)+I::footprint();
112       }
113       bool is_valid() const { return (this != 0) && (*subject_).is_valid(); }
114       bool is_root() const { return (*subject_).is_root(); }
115       bool is_leaf() const { return (*subject_).is_leaf(); }
116       bool is_marked() const { return (*subject_).is_marked(); }
117       height_type height() const { return (*subject_).height(); }
118
119       void* owner() const { return (*subject_).owner(); }
120       void*& owner() { return (*subject_).owner(); }
121
122       PE* root() const {
123         I* i(static_cast<I*>((*subject_).root()));
124         return i == 0?0:(*i).element().prx_ptr;
125       }
126
127       PE* most_significant_child() const {
128         // Desc: locate weaker subtree before an actual split.
129         I* i (static_cast<I*>((*subject_).most_significant_child()));
130         return i == 0?0:(*i).element().prx_ptr;
131       }
132
133       PE* distinguished_ancestor() const {
134         I* i(static_cast<I*>((*subject_).distinguished_ancestor()));
135         return i == 0?0:(*i).element().prx_ptr;
136       }
137
```

```
138         PE* distinguished_descendant() const {
139           I* i(static_cast<I*>((*subject_).distinguished_descendant()));
140           return i == 0?0:(*i).element().prx_ptr;
141         }
142
143         PE const* successor() const {
144           I* i(static_cast<I*>((*subject_).successor()));
145           return i == 0?0:(*i).element().prx_ptr;
146         }
147         PE * successor() {
148           I* i(static_cast<I*>((*subject_).successor()));
149           return i == 0?0:(*i).element().prx_ptr;
150         }
151
152         bool tree_valid() const { return (*subject_).tree_valid(); }
153
154         // operations.
155         PE* join(PE* weaker) { return (*(*subject_).join((*weaker).subject_)).element().prx_ptr;
                }
156
157         template <typename C2>
158         PE* join(PE* other, C2 const& comparator) {
159           if (!comparator((*this).element(), (*other).element()))
160             return join(other);
161           else return (*other).join(this);
162         }
163
164         PE* split() { return (* static_cast<I*>((*subject_).split())).element().prx_ptr; }
165
166         template <class Boolean>
167         PE* construct(size_type& n, const Boolean calc_size) {
168           I* i(static_cast<I*>((*subject_).construct(n, calc_size)));
169           return i == 0?0:(*i).element().prx_ptr;
170         }
171
172         PE* release_root() {
173           I* i(static_cast<I*>((*subject_).release_root()));
174           return i == 0?0:(*i).element().prx_ptr;
175         }
176
177         PE* release_subheap() {
178           I* i(static_cast<I*>((*subject_).release_subheap()));
179           return i == 0?0:(*i).element().prx_ptr;
180         }
181
182         PE* next_root_get() {
183           I* i(static_cast<I*>((*subject_).owner()));
184           return i == 0?0:(*i).element().prx_ptr;
185         }
186
187         void next_root_set(PE* r) {
188           (*subject_).owner() = (r == 0?0:(*r).subject_);
189         }
190
191         hole_type splice_out(bool forced = false)
192         { return (*subject_).splice_out(forced); }
193
194         template <typename H>
195         void splice_in(hole_type& hole, H&, bool forced = false) {
196           int dummy; (*subject_).splice_in(hole, dummy, forced);
197         }
198
199         PE* swap(PE* p) {
200           // general swap not (yet) supported.
201           assert(p == (*this).distinguished_ancestor());
202           return promote(p);
203         }
204
205         void swap_roots(PE* p) {
206           assert(is_valid()); assert((*p).is_valid());
207           //return (
208           //  * static_cast<I*>((*subject_).swap_roots((*p).subject_))
209           //  ).element().prx_ptr;
210           (*subject_).swap_roots((*p).subject_);
```

```
211         }
212
213         PE* promote(PE* p) {
214           assert(is_valid()); assert((*p).is_valid());
215           assert(p == (*this).distinguished_ancestor());
216           return (
217             * static_cast<I*>((*subject_).promote((*p).subject_))
218             ).element().prx_ptr;
219         }
220
221         PE* promote_alt(PE* p) {
222           assert(is_valid()); assert((*p).is_valid());
223           assert(p == (*this).distinguished_ancestor());
224           return (
225             * static_cast<I*>((*subject_).promote_alt((*p).subject_))
226             ).element().prx_ptr;
227         }
228
229         std::string str() const { return (*subject_).str(); }
230       };
231
232       template <typename B, typename V, typename C, typename A>
233       typename proxy<B,V,C,A>::allocator_type proxy<B,V,C,A>::allo_ = allocator_type();
234
235       template <typename B, typename V, typename C, typename A>
236       std::ostream& operator<<
237         (std::ostream &os, proxy_subject_element_type<B,V,C,A> const& c) {
238         os << "." << c.value;
239         return os;
240       }
241     }
242   }
243   #endif
```

A.23 node_with_direct_value.hpp

```
1   #ifndef _CPHSTL_PQFW_NODE_WITH_VALUE_HPP_
2   #define _CPHSTL_PQFW_NODE_WITH_VALUE_HPP_
3
4   /*
5   Desc: Node with contained value.
6   Exception safety of wrapped methods unchanged.
7   Auth: Jyrki Katajainen, Asger Bruun © 2009
8   Stat: 20091008 Tested with ms-compiler & gcc 4.3.3.
9
10  *     -> (n)[v]
11  */
12
13  #include <string> // std::string
14  #include <iostream> // std::ostream
15  #include <sstream> // std::sstream
16  #include <memory> // std::allocator
17  #include <functional> // less
18
19  #include "pwh_node.hpp"
20
21  namespace cphstl {
22    namespace pqfw_node {
23
24      template <int number_of_bits> struct size_traits {};
25
26      template <> struct size_traits<32>  { // ?? unsigned
27         typedef char height_type;
28      };
29      template <> struct size_traits<64>  {
30         typedef char height_type;
31      };
32      template <> struct size_traits<128>  {
33         typedef unsigned char height_type;
34      };
35
36      //20100730 add begin
37      // Make class independant from physical memory.
```

```
38      template <typename Derived
39        , typename A = std::allocator<Derived>
40      >
41      class using_allocator {
42      public:
43        typedef typename A::template rebind<Derived>::other allocator_type;
44        typedef typename allocator_type::pointer pointer;
45        typedef typename allocator_type::const_pointer const_pointer;
46        typedef typename allocator_type::reference reference;
47        typedef typename allocator_type::const_reference const_reference;
48        typedef typename allocator_type::size_type size_type; //!!!
49        typedef typename size_traits<sizeof(size_type)*8>::height_type height_type;
50      };
51      //20100730 add end !!!
52
53      template <typename V = int
54        , typename C = std::less<V>
55        , typename A = std::allocator<V>
56      >
57      class value_node
58        : public using_allocator<value_node<V,C,A>,A> //20100730 add
59      {
60        V value_;
61        value_node();
62        value_node(value_node const&);
63        value_node& operator = (value_node const&);
64      public:
65        typedef V value_type;
66        typedef C comparator_type;
67
68        value_node(V const& value) : value_(value)  {}
69
70        V const& element() const { return value_;  }
71        V& element() { return value_; }
72      };
73
74      template <
75        typename DB = pwh_node_base
76        , typename V = int
77        , typename C = std::less<V>
78        , typename A = std::allocator<V>
79      >
80      class w_direct_value : public DB, public value_node<V,C,A> {
81        typedef w_direct_value<DB,V,C,A> DE;
82
83        static C compare_;
84
85        w_direct_value();
86        w_direct_value(w_direct_value const&);
87        w_direct_value& operator = (w_direct_value const&);
88
89      public:
90
91        static const bool promote_is_root_list_neutral = false;
92
93        typedef V value_type;
94        typedef C comparator_type;
95        typedef A allocator_type;
96        typedef typename value_node<V,C,A>::size_type size_type;
97        typedef typename value_node<V,C,A>::height_type height_type;
98        typedef typename DB::hole_type hole_type;
99        typedef DB node_base_type;
100
101       template<template <typename T> class U>
102       struct wrap_allocator {
103         typedef w_direct_value<DB,V,C,U<A> > other;
104       };
105
106       template<typename A5>
107       struct replace_allocator {
108         typedef w_direct_value<DB,V,C,A5> other;
109       };
110
111       template<typename V2>
```

```
112         struct rebind_value_type {
113           typedef w_direct_value<DB, V2, C, A> other;
114         };
115 
116         w_direct_value(V const& v)
117           : DB(), value_node<V,C,A>(v) {}
118 
119         static typename DB::size_type footprint() { return sizeof(DE); }
120 
121         DE* next_root_get() {
122           return (DE*) DB::owner();
123         }
124         void next_root_set(DE* r) {
125           DB::owner() = r;
126         }
127 
128         DE* join(DE* weaker) {
129           return static_cast<DE*>(DB::join(weaker));
130         }
131 
132         template <typename OC>
133         DE* join(DE* other, OC const& comparator) {
134           if (!comparator((*this).element(), (*other).element()))
135             return join(other);
136           else return (*other).join(this);
137         }
138 
139         std::string str() const { return (*this).str<DE>(); }
140 
141       protected:
142 
143         template <typename D>
144         std::string str() const {
145           if(this == 0) return DB::template str<D>();
146           else {
147             std::ostringstream os;
148             os << (*this).element();
149             os << DB::template str<D>();
150             return os.str();
151           }
152         }
153 
154       };
155       template <typename DB, typename V, typename C, typename A>
156       typename w_direct_value<DB,V,C,A>::comparator_type w_direct_value<DB,V,C,A>::compare_ = C
              ();
157     } // namespace pqfw_node
158   } // namespace chpstl
159   #endif
```

A.24 node_with_facade.hpp

```
1   #ifndef _CPHSTL_PQFW_NODE_WITH_FACADE_HPP_
2   #define _CPHSTL_PQFW_NODE_WITH_FACADE_HPP_
3 
4   /*
5   Desc: Makes the node compatible with priority-queue-framework,
6   :     makes the node internals private and downcasts everything.
7   Auth: Jyrki Katajainen, Asger Bruun © 2009
8   Stat: 20091008 Tested with ms-compiler & gcc 4.3.3.
9   Prob: There are problems with the generic downcast here and there
10        (the places with static cast).
11  */
12 
13  #include <memory> // std:allocator
14  #include "root_list.hpp"
15  #include "has_member.hpp"
16 
17 
18  // Define compile time type tests:
19  //   bool cphstl::has_member_root_list_type<T>::value
20  //   bool cphstl::has_member_prev_root<T>::value
21  DEFINE_STATIC_TEST_HAS_MEMBER(root_list_type)
```

```
22  DEFINE_STATIC_TEST_HAS_MEMBER(prev_root)
23
24
25  namespace cphstl {
26    namespace pqfw_node {
27      template <
28        typename B1/*ase*/
29      >
30      class with_facade: public B1 {
31        with_facade();
32        with_facade(with_facade const&);
33        with_facade& operator = (const with_facade&);
34
35        template <bool has_owner, class B2 = void> struct owner_dispatch{
36          // dummy methods for (2-ptr) node bases not supporting owner.
37          static void* owner(B2 const* /*b*/) { assert(0); return 0; }
38          static void*& owner(B2 * b) { assert(0); return (void*&) (*b); }
39          static B2* release_root(B2*) { assert(0); return 0; }
40          static B2* release_subheap(B2*) { assert(0); return 0; }
41        };
42        template <class B2> struct owner_dispatch<true, B2> {
43          // CPH STL owner-attribute dependant methods.
44          static void* owner(B2 const* p) { return (*p).owner(); }
45          static void*& owner(B2 * p) { return (*p).owner(); }
46
47          static B2* release_root(B2 * p) {
48            return static_cast<B2*>((*p).release_root());
49          }
50          static B2* release_subheap(B2 * p) {
51            return static_cast<B2*>((*p).release_subheap());
52          }
53        };
54
55        template <bool special_list, class D2 = void>
56        struct root_list_type_select {
57          // type for node bases with simple root_list_type.
58          typedef root_list<D2, false> type;
59        };
60        template <class D2> struct root_list_type_select <true, D2> {
61          typedef typename D2::root_list_type::template
62            rebind_encapsulator_type<D2>::other type;
63        };
64
65
66        typedef owner_dispatch<node_traits<
67          typename B1::node_base_type>::has_owner, B1
68        > od;
69
70        typedef with_facade<B1> D; // derived if crtp.
71
72
73      public:
74        // ----- direct binary and priority queue framework common -----
75
76        // types.
77
78        static const bool promote_is_root_list_neutral
79         = B1::promote_is_root_list_neutral;
80
81        typedef typename B1::value_type value_type;
82        typedef typename B1::comparator_type comparator_type;
83        typedef typename B1::allocator_type allocator_type;
84        typedef typename B1::allocator_type::template
85          rebind<D>::other::pointer pointer;
86        typedef typename B1::allocator_type::template
87          rebind<D>::other::const_pointer const_pointer;
88        typedef typename B1::allocator_type::template
89          rebind<D>::other::reference reference;
90        typedef typename B1::allocator_type::template
91          rebind<D>::other::const_reference const_reference;
92
93        typedef typename B1::allocator_type::template
94          rebind<B1>::other::pointer base_pointer;
95
```

```
96          typedef typename B1::size_type size_type;
97          typedef typename B1::height_type height_type;
98          typedef typename B1::hole_type hole_type;
99
100         typedef typename root_list_type_select<
101           cphstl::has_member_root_list_type<B1>::value, D
102         >::type root_list_type;
103
104
105         typedef typename B1::node_base_type node_base_type;
106
107         // properties.
108
109         static size_type footprint() { return B1::footprint(); }
110
111         bool is_valid() const { return B1::is_valid(); }
112         bool is_root() const { return B1::is_root(); }
113         bool is_leaf() const { return B1::is_leaf(); }
114
115         value_type const& element() const { return B1::element(); }
116         value_type& element() { return B1::element(); }
117
118         const_pointer distinguished_ancestor() const {
119           return static_cast<const_pointer>(B1::distinguished_ancestor());
120         }
121         pointer distinguished_ancestor() {
122           return static_cast<pointer>(B1::distinguished_ancestor());
123         }
124
125         pointer successor() { return static_cast<D *>(B1::successor()); }
126         const_pointer successor() const { return static_cast<const_pointer>(B1::successor());
                }
127
128         // operations.
129
130         explicit with_facade(value_type const& v) : B1(v) {}
131
132         template <typename C, typename M>
133         pointer join(pointer q, C const& comparator, M&) {
134           return join(q, comparator);
135         }
136
137         template <typename C>
138         pointer join(pointer other, C const& comparator) {
139           pointer ret(downcast(B1::join(other, comparator)));
140           assert(ret == this || (*this).element() != (*other).element());
141           return ret;
142         }
143         pointer join(pointer weaker) { return downcast(B1::join(weaker)); }
144
145         pointer split() { return static_cast<pointer>(B1::split()); }
146
147         pointer promote(pointer q) {
148           assert((*this).is_valid()); assert((*q).is_valid());
149           assert((*this).distinguished_ancestor() == q);
150           pointer res(static_cast<pointer>(B1::promote(q)));
151           assert((*this).is_valid()); assert((*q).is_valid());
152           assert((*q).distinguished_ancestor() == this);
153           return res;
154         }
155
156         pointer promote_alt(pointer q) {
157           assert((*this).is_valid()); assert((*q).is_valid());
158           assert((*this).distinguished_ancestor() == q);
159           pointer res(static_cast<pointer>(B1::promote_alt(q)));
160           assert((*this).is_valid()); assert((*q).is_valid());
161           assert((*q).distinguished_ancestor() == this);
162           return res;
163         }
164
165         // ----- direct binary heap store special -----
166
167         // type redefinitions.
168
```

```
169         template<typename A5>
170         struct replace_allocator {
171           typedef typename B1::template replace_allocator<A5>::other new_base;
172           typedef with_facade<new_base> other;
173         };
174
175         // properties.
176
177         height_type height() const { return B1::height(); }
178
179         pointer most_significant_child() {
180           return static_cast<pointer>(B1::most_significant_child());
181         }
182
183         std::string str() const { return B1::str(); }
184
185         // operations.
186
187         template <class Boolean>
188         pointer construct(size_type& n, const Boolean calc_size) {
189           return static_cast<pointer>(B1::construct(n, calc_size));
190         }
191
192         // forward root list ability.
193
194         pointer next_root_get() {
195           return (pointer) B1::next_root_get();
196         }
197         void next_root_set(pointer r) {
198           B1::next_root_set(r);
199         }
200
201         // optional backward root list ability.
202
203         template <bool has_prev_root, class B2 = void>
204         struct prev_root_dispatch{ // unidir dummy methods.
205           static bool prev_rootable(const_pointer) {
206             // Note: abused on unidir version for faster !is_leaf test.
207             return true;
208           }
209           static pointer prev_root_get(pointer) { assert(0); return 0; }
210           static void prev_root_set(pointer, pointer) { assert(0); }
211         };
212         template <class B2>
213         struct prev_root_dispatch<true, B2> { // real methods.
214           static bool prev_rootable(const_pointer t) {
215             return (*t).B1::prev_rootable();
216           }
217           static pointer prev_root_get(pointer t) {
218             return (pointer) (*t).B1::prev_root();
219           }
220           static void prev_root_set(pointer t, pointer r) {
221             (*t).prev_root() = r;
222           }
223         };
224         typedef prev_root_dispatch<
225           has_member_prev_root<B1>::value, B1
226         > prd;
227
228         bool prev_rootable() const {
229           return prd::prev_rootable(this);
230         }
231         pointer prev_root_get() {
232           return (pointer) prd::prev_root_get(this);
233         }
234         void prev_root_set(pointer r) {
235           prd::prev_root_set(this,r);
236         }
237
238         // node iterator compatibility.
239
240         value_type const& content() const { return B1::element();  }
241         value_type& content() { return B1::element();  }
242
```

```
243         // heap clear.
244
245         template<typename PA // protected access
246         > static void clear_heap(pointer h, PA& pa) {
247           // assert(heap is ejected from list)
248           assert(h != 0); assert((*h).is_root());
249           pointer p(h);
250           if((*p).is_leaf()) {
251             pa.destroy(p);
252           } else {
253             pointer q((*p).split());
254             clear_heap(p,pa);
255             clear_heap(q,pa);
256           }
257         }
258
259         // ----- priority queue framework compatibility -----
260
261         with_facade(value_type const& v, allocator_type const& /*ignored*/) : B1(v) {}
262
263         void* owner() const { return od::owner(this); }
264         void*& owner() { return od::owner(this); }
265
266         pointer release_root() {
267           // Desc: Vuillemin construct with reverse root list ordering.
268           return static_cast<pointer>(od::release_root(this));
269         }
270
271         template <typename M>
272         pointer release_subheap(M& /*ignored*/) {
273           // Desc: disconnect reverse root list head from tail.
274           return static_cast<pointer>(od::release_subheap(this));
275         }
276         pointer release_subheap() {
277           // Desc: disconnect reverse root list head from tail.
278           return static_cast<pointer>(od::release_subheap(this));
279         }
280
281
282         template <typename C, typename M>
283         bool is_valid(C const& /*ignored*/, M const& /*ignored*/) const {
284           return B1::is_valid();
285         }
286
287         template <typename C>
288         const_pointer distinguished_descendant(C const& /*ignored*/) const {
289           return static_cast<pointer>(B1::distinguished_descendant());
290         }
291         template <typename C>
292         pointer distinguished_descendant(C const& /*ignored*/) {
293           return static_cast<pointer>(B1::distinguished_descendant());
294         }
295
296         template <typename C>
297         pointer split(C const& /*ignored*/) { return (*this).split(); }
298
299         template <typename C, typename M>
300         pointer split(C const& /*ignored*/, M& mark_store) {
301           mark_store.unmark((*this).most_significant_child());
302           return (*this).split();
303         }
304
305         template <typename C>
306         pointer fast_join(pointer q, pointer, C const& comparator) {
307           return join(q, comparator);
308         }
309         template <typename C>
310         pointer fast_split(C const& /*ignored*/) {
311           return static_cast<pointer>(B1::split());
312         }
313
314         hole_type splice_out(bool forced = false) { return B1::splice_out(forced); }
315
316         template <typename H>
```

```
317         void splice_in(hole_type& hole, H& heap_store, bool forced = false) {
318           B1::splice_in(hole, heap_store, forced);
319           if ((*this).is_root()) {
320             typedef typename H::heap_proxy_type PT;
321             typedef typename PT::encapsulator_type ET;
322             //PT* p((PT*) owner());
323             assert((*this).is_valid());
324             // !!! (*p).update(static_cast<pointer>(this));
325             heap_store.replace(this);
326             assert((*this).is_valid());
327           }
328         }
329 
330         void splice_in(hole_type h,bool forced = false)
331         { B1::splice_in(h, forced); }
332 
333         pointer swap(pointer q) { return downcast(B1::swap(q)); }
334         pointer swap_roots(pointer q) {
335           B1::swap_roots(q);
336           return this;
337         }
338 
339         void show_tree() const { std::cout << str(); }
340 
341         template <typename C>
342         bool tree_valid(C const& /*ignored*/) const { return B1::is_valid(); }
343 
344         template <typename C, typename M>
345         bool tree_valid(C const& /*ignored*/, M const& /*ignored*/) const { return B1::
              is_valid(); }
346 
347 
348         // ----- priority queue framework compatibility maybe not in use -----
349 
350         bool is_marked() const { return B1::is_marked(); }
351 
352         pointer root() { return static_cast<pointer>(B1::root()); }
353 
354         const_pointer root() const {
355           return static_cast<const_pointer>(
356             B1::root()
357           );
358         }
359 
360         // -----
361 
362       private:
363         const_pointer const downcast(B1 const* const b) const {
364           return static_cast<const_pointer const>(b);
365         }
366         pointer const downcast(base_pointer const b) const {
367           return static_cast<pointer const>(b);
368         }
369         pointer const downcast(base_pointer const b) {
370           return static_cast<pointer const>(b);
371         }
372 
373         friend std::ostream & operator << (std::ostream & os, const_reference e) {
374           os << e.str();
375           return os;
376         }
377       };
378   #define w_facade with_facade
379     }
380   }
381   #endif
```

A.25 node_with_indirect_value.hpp

```
1   #ifndef _CPHSTL_PQFW_NODE_WITH_INDIRECT_VALUE_HPP_
2   #define _CPHSTL_PQFW_NODE_WITH_INDIRECT_VALUE_HPP_
3 
4   /*
```

```
5  Desc: Node with indirect value
6        (actually a value with indirect node).
7  Feat: Node swap in constant time.
8        Exception safety of wrapped methods unchanged.
9  Auth: Jyrki Katajainen, Asger Bruun © 2009
10 Note: Two versions, depends on the design of the node base.
11
12 *     -> [v]
13 *          |
14 *         (n)
15 */
16
17 #include <string> // std::string
18 #include <iostream> // std::ostream
19 #include <memory> // std::allocator
20
21 #include "light_binomial_node.hpp"
22 #include "node_with_direct_value.hpp"
23 #include "node_with_facade.hpp"
24
25 namespace cphstl {
26   namespace pqfw_node {
27     template < typename B = light_binomial_node_base
28       , typename V = int
29       , typename C = std::less<V>
30       , typename A = std::allocator<V>
31     >
32     class w_indirect_value  {
33       typedef w_indirect_value<B,V,C,A> IE;
34       typedef w_direct_value<B, IE* /*, AT*/> I;
35       V value_;
36       I* node_; // the node structure layer.
37
38       w_indirect_value();
39       w_indirect_value(w_indirect_value const&);
40       w_indirect_value& operator = (w_indirect_value const&);
41
42     public:
43       static const bool promote_is_root_list_neutral = true;
44
45       typedef B node_base;
46       typedef V value_type;
47       typedef C comparator_type;
48       typedef typename A::template rebind<I>::other allocator_type;
49       typedef typename I::size_type size_type;
50       typedef typename I::height_type height_type;
51       typedef B node_base_type;
52       // typedef typename B::root_list_type root_list_type;
53
54       struct hole_type : public I::hole_type {
55         hole_type(IE* n, bool auto_splice_out, bool forced = false)
56           : I::hole_type((*n).node_, auto_splice_out, forced) {}
57         hole_type(typename I::hole_type const& h) : I::hole_type(h) {}
58       };
59
60     private:
61
62       static allocator_type allo_;
63
64     public:
65
66       w_indirect_value(V const& v) : value_(v) { // node_ = new I(this);
67         node_ = allo_.allocate(1);
68         try { new(node_) I(this); }
69         catch (...) { allo_.deallocate(node_, 1); throw; }
70       }
71       ~w_indirect_value() { // delete(node_);
72         (*node_).~I();
73         allo_.deallocate(node_, 1);
74       }
75
76       V const& element() const { return value_;  }
77       V& element() { return value_; }
78
```

```
79        // properties.
80        static size_type footprint() {
81          return sizeof(IE)+I::footprint();
82        }
83        bool is_valid() const { return (*node_).is_valid(); }
84        bool is_root() const { return (*node_).is_root(); }
85        bool is_leaf() const { return (*node_).is_leaf(); }
86        bool is_marked() const { return (*node_).is_marked(); }
87        height_type height() const { return (*node_).height(); }
88
89        void* owner() const { return (*node_).owner(); }
90        void*& owner() { return (*node_).owner(); }
91
92        IE* root() const {
93          I* i(static_cast<I*>((*node_).root()));
94          return i == 0?0:(*i).element();
95        }
96
97        IE* distinguished_ancestor() const {
98          I* i(static_cast<I*>((*node_).distinguished_ancestor()));
99          return i == 0?0:(*i).element();
100       }
101       IE* distinguished_descendant() const {
102         I* i(static_cast<I*>((*node_).distinguished_descendant()));
103         return i == 0?0:(*i).element();
104       }
105
106       IE const* successor() const {
107         I* i(static_cast<I*>((*node_).successor()));
108         return i == 0?0:(*i).element();
109       }
110       IE * successor() {
111         I* i(static_cast<I*>((*node_).successor()));
112         return i == 0?0:(*i).element();
113       }
114
115       bool tree_valid() const { return (*node_).tree_valid(); }
116
117       // operations.
118       IE* join(IE* weaker) { return (*(*node_).join((*weaker).node_)).element(); }
119       template <typename OC>
120       IE* join(IE* other, OC const& comparator) {
121         if (!comparator((*this).element(), (*other).element()))
122           return join(other);
123         else return (*other).join(this);
124       }
125
126       IE* most_significant_child() {
127         // Desc: locate weaker subtree before an actual split.
128         I* i(static_cast<I*>((*node_).most_significant_child()));
129         return i == 0?0:(*i).element();
130       }
131       IE* split() { return (* static_cast<I*>((*node_).split())).element(); }
132
133       template <class Boolean>
134       IE* construct(size_type& n, const Boolean calc_size) {
135         // convert to indirect (and waste time)
136         I* i(static_cast<I*>((*node_).construct(n, calc_size)));
137         return i == 0?0:(*i).element();
138       }
139
140       IE* release_root() {
141         I* i(static_cast<I*>((*node_).release_root()));
142         return i == 0?0:(*i).element();
143       }
144
145       IE* release_subheap() {
146         I* i(static_cast<I*>((*node_).release_subheap()));
147         return i == 0?0:(*i).element();
148       }
149
150       IE* next_root_get() {
151         I* i(static_cast<I*>((*node_).owner()));
152         return i == 0?0:(*i).element();
```

```
153         }
154         void next_root_set(IE* r) {
155           (*node_).owner() = (r == 0?0:(*r).node_);
156         }
157
158         hole_type splice_out(bool forced = false)
159         { return (*node_).splice_out(forced); }
160
161         template <typename H>
162         void splice_in(hole_type& hole, H&, bool forced = false)
163         { int dummy; (*node_).splice_in(hole, dummy, forced); }
164
165         IE*  swap(IE* p) {
166           // Desc: swap node positions. T(n): O(1). ES: no throw.
167           //       operation between different tress are allowed.
168           assert(is_valid()); assert((*p).is_valid());
169           std::swap((*(*this).node_).element() , (*(*p).node_).element());
170           std::swap((*this).node_ , (*p).node_);
171           return (*(*this).node_).element();
172         }
173
174         IE* swap_roots(IE* p) {
175           assert(is_valid()); assert((*p).is_valid());
176           assert(is_root()); assert((*p).is_root());
177           return swap(p);
178         }
179
180
181         IE* promote(IE* p) {
182           assert(is_valid()); assert((*p).is_valid());
183           assert(p == (*this).distinguished_ancestor());
184           return swap(p);
185         }
186
187         IE* promote_alt(IE* p) {
188           assert(is_valid()); assert((*p).is_valid());
189           assert(p == (*this).distinguished_ancestor());
190           return (*swap(p)).distinguished_ancestor();
191         }
192
193         std::string str() const { return (*node_).str(); }
194       };
195
196       template <typename B, typename V, typename C, typename A>
197       typename w_indirect_value<B,V,C,A>::allocator_type
198         w_indirect_value<B,V,C,A>::allo_ = allocator_type();
199
200       template <typename B, typename V>
201       std::ostream& operator<< (std::ostream &os, w_indirect_value<B,V>*iv) {
202         os << "" << (*iv).element() << ".";
203         return os;
204       }
205     }
206   }
207
208   ////////////////////////////////////////////////////////////////
209   // Desc: alternate solution.
210
211   namespace cphstl {
212     namespace pqfw_node {
213       template <
214         typename B = w_direct_value<light_binomial_node_base, char>
215       >
216       class w_indirection : public value_node<typename B::value_type
217         , typename B::comparator_type, typename B::allocator_type>
218       {
219         typedef w_indirection<B> IE;
220         typedef typename B::template rebind_value_type<IE*>::other I;
221
222         I* node_; // the node structure layer.
223
224         w_indirection();
225         w_indirection(w_indirection const&);
226         w_indirection& operator = (w_indirection const&);
```

```
227
228     public:
229
230       static const bool promote_is_root_list_neutral = true;
231       typedef typename B::value_type value_type;
232       typedef typename B::allocator_type::template rebind<I>::other allocator_type;
233       typedef typename I::size_type size_type;
234       typedef typename I::height_type height_type;
235       typedef B node_base_type;
236
237       struct hole_type : public I::hole_type {
238         hole_type(IE* n) : I::hole_type((*n).node_) {}
239         hole_type(typename I::hole_type const& h) : I::hole_type(h) {}
240       };
241
242
243
244     private:
245
246       static allocator_type allo_;
247
248     public:
249
250       w_indirection(value_type const& v)
251         : value_node<typename B::value_type, typename B::comparator_type, typename B::
              allocator_type>
252         (v)
253       {
254         node_ = allo_.allocate(1);
255         try { new(node_) I(this); }
256         catch (...) { allo_.deallocate(node_, 1); throw; }
257       }
258       ~w_indirection() { // delete(node_);
259         (*node_).~I();
260         allo_.deallocate(node_, 1);
261       }
262
263       // properties.
264       static typename B::size_type footprint() {
265         return sizeof(IE)+I::footprint();
266       }
267       bool is_valid() const { return (*node_).is_valid(); }
268       bool is_root() const { return (*node_).is_root(); }
269       bool is_leaf() const { return (*node_).is_leaf(); }
270       bool is_marked() const { return (*node_).is_marked(); }
271       height_type height() const { return (*node_).height(); }
272
273       void* owner() const { return (*node_).owner(); }
274       void*& owner() { return (*node_).owner(); }
275
276       IE* root() const {
277         I* i(static_cast<I*>((*node_).root()));
278         return i == 0?0:(*i).element();
279       }
280
281       IE* distinguished_ancestor() const {
282         I* i(static_cast<I*>((*node_).distinguished_ancestor()));
283         return i == 0?0:(*i).element();
284       }
285       IE* distinguished_descendant() const {
286         I* i(static_cast<I*>((*node_).distinguished_descendant()));
287         return i == 0?0:(*i).element();
288       }
289
290       IE const* successor() const {
291         I* i(static_cast<I*>((*node_).successor()));
292         return i == 0?0:(*i).element();
293       }
294       IE * successor() {
295         I* i(static_cast<I*>((*node_).successor()));
296         return i == 0?0:(*i).element();
297       }
298
299       bool tree_valid() const { return (*node_).tree_valid(); }
```

```
300
301         // operations.
302         IE* join(IE* weaker) { return (*(*node_).join((*weaker).node_)).element(); }
303         template <typename OC>
304         IE* join(IE* other, OC const& comparator) {
305           if (!comparator((*this).element(), (*other).element()))
306             return join(other);
307           else return (*other).join(this);
308         }
309
310         IE* most_significant_child() {
311           // Desc: locate weaker subtree before an actual split.
312           I* i(static_cast<I*>((*node_).most_significant_child()));
313           return i == 0?0:(*i).element();
314         }
315         IE* split() { return (* static_cast<I*>((*node_).split())).element(); }
316
317         template <class Boolean>
318         IE* construct(size_type& n, const Boolean calc_size) {
319           I* i(static_cast<I*>((*node_).construct(n, calc_size)));
320           return i == 0?0:(*i).element();
321         }
322
323         IE* next_root_get() {
324           I* i(static_cast<I*>((*node_).next_root_get()));
325           return i == 0?0:(*i).element();
326         }
327         void next_root_set(IE* r) {
328           (*node_).next_root_set(r == 0?0:(*r).node_);
329         }
330
331         hole_type splice_out(bool forced = false)
332         { return (*node_).splice_out(forced); }
333
334         int dummy;
335         template <typename H>
336         void splice_in(hole_type& hole, H&, bool forced = false)
337         { (*node_).splice_in(hole, dummy, forced); }
338
339         IE*  swap(IE* p) {
340           // Desc: swap node positions. T(n): O(1). ES: no throw.
341           //       operation between different tress are allowed.
342           assert(is_valid()); assert((*p).is_valid());
343           std::swap((*(*this).node_).element() , (*(*p).node_).element());
344           std::swap((*this).node_ , (*p).node_);
345           return (*(*this).node_).element();
346         }
347
348         IE* swap_roots(IE* p) {
349           assert(is_valid()); assert((*p).is_valid());
350           assert(is_root()); assert((*p).is_root());
351           return swap(p);
352         }
353
354
355         IE* promote(IE* p) {
356           assert(is_valid()); assert((*p).is_valid());
357           assert(p == (*this).distinguished_ancestor());
358           return swap(p);
359         }
360
361         std::string str() const { return (*node_).str(); }
362       };
363
364       template <typename B>
365       typename w_indirection<B>::allocator_type w_indirection<B>::allo_ = allocator_type();
366
367       template <typename B>
368       std::ostream& operator<< (std::ostream &os, w_indirection<B>*iv) {
369         os << "" << (*iv).element() << ".";
370         return os;
371       }
372     }
373   }
```

```
374
375 #endif
```

A.26 priority_queue_framework_for_dbhs.hpp

```
1  #ifndef _CPHSTL_PRIORITY_QUEUE_FRAMEWORK_FOR_dhs_
2  #define _CPHSTL_PRIORITY_QUEUE_FRAMEWORK_FOR_dhs_
3
4  /*
5  Desc: Makes the priority_queue_framework compatible with the
6  :     direct_heap_store. The major problem was that the
7  :     pqfw extract(p) has been breaking the pre-conditions for
8  :     the nodes of dhs for the last months and that the
9  :     pqfw therefore had to run in a modified form.
10 :     Now, the problem is wrapped into a minimal addition.
11 Auth: Asger Bruun 2009-2010
12 */
13
14 #include "multiple-heap-framework.h++"
15 #include "proxy-list-heap-store.h++"
16
17 namespace cphstl {
18
19   template <
20     typename V,
21     typename C = std::less<V>,
22     typename A = std::allocator<V>,
23     typename E = weak_heap_node<V, A>,
24     typename P = heap_proxy<E>,
25     typename H = proxy_list_heap_store<C, A, E, P>,
26     typename M = blank_mark_store<C, A, E>
27   >
28   class pqfw_for_dhs_nodes
29     : public multiple_heap_framework<V,C,A,E,H,M> {
30
31       typedef multiple_heap_framework<V,C,A,E,H,M> B;
32       typedef typename B::size_type size_type;
33
34     explicit pqfw_for_dhs_nodes(pqfw_for_dhs_nodes const&); // undefined
35
36   public:
37
38     enum { number_system = 1 };
39
40     explicit pqfw_for_dhs_nodes(
41       C const& c = C(), A const& a = A()
42       ) : multiple_heap_framework<V,C,A,E,H,M>(c,a) {}
43
44     E* extract() { return B::extract(); }
45
46     bool empty() const {
47       return (*this).size() == size_type(0);
48     }
49
50
51     void extract(E* p) {
52       assert(p != 0);
53       E* replacement = extract();
54       assert(replacement != 0);
55       if (p == replacement) {
56         return;
57       }
58       (*this).mark_store.unmark(p);
59       typename E::hole_type hole_at_p = (*p).splice_out(true);
60
61       std::list<E*, A> special_path;
62       if(true || careful) while (!(*p).is_leaf()) { //ab +
63         (*this).mark_store.unmark((*p).most_significant_child());
64         assert((*this).mark_store.is_valid());
65         special_path.push_front((*p).split(B::comparator, B::mark_store));
66       } else {
67         E* q = (*p).distinguished_descendant(B::comparator);
68         while (q != 0) {
```

```
69              (*this).mark_store.unmark(q);
70              assert((*this).mark_store.is_valid());
71              special_path.push_front(q);
72              q = (*q).distinguished_descendant(B::comparator);
73            }
74          }
75
76          E* r = replacement;
77          while (! special_path.empty()) {
78            E* s = special_path.front();
79            r = (*r).fast_join(s, replacement, B::comparator);
80            special_path.pop_front();
81          }
82
83          (*r).splice_in(hole_at_p, (*this).heap_store, true);
84
85          (*this).mark_store.mark(r);
86          (*this).mark_store.reduce((*this).heap_store);
87          if (p == (*this).top_) {
88            (*this).top_ = (*this).heap_store.find_top();
89            E* t = (*this).mark_store.find_top();
90            if (t != 0 && (*this).comparator((*((*this).top_)).element(), (*t).element())) {
91              (*this).top_ = t;
92            }
93          }
94        }
95      };
96    }
97    #endif
```

A.27 pwh_node.hpp

```
1   /*
2     A weak-heap node - first adapted, then modified.
3
4     Author: Jyrki Katajainen © 2009
5
6   Desc: Protection against the changing interface of the PQFW perfect weak heap node.
7   Every rewrite is expensive, especially because of the many external inherited variants.
8   Giving up tracking the moving target, we will have to do with a snapshot of 20091025.
9   Rewrite: Asger.
10  */
11
12
13  #ifndef _CPHSTL_PWH_NODE_
14  #define _CPHSTL_PWH_NODE_
15
16  #include "assert.h++"
17  #include <cstddef> // std::size_t
18  #include <iostream>
19  #include <list>
20
21  #include "root_list.hpp"
22
23  namespace cphstl {
24    namespace pqfw_node {
25      class pwh_node_base {
26        typedef pwh_node_base N;
27
28        union {
29          void* owner_;
30          N* left_;
31        };
32        N* parent_;
33        N* right_;
34
35        pwh_node_base(pwh_node_base const&);
36        pwh_node_base& operator = (pwh_node_base const&);
37
38      public:
39
40        typedef std::size_t size_type;
41        typedef unsigned char height_type;
```

```
42
43        struct hole_type {
44          N* da;
45          N* parent;
46          N* current;
47          union {
48            N* left;
49            void* owner;
50          };
51
52          hole_type(N* p, bool auto_splice_out, bool forced = false)
53            : da((*p).distinguished_ancestor()), parent((*p).parent()), current(p)
54          {
55            if (parent != 0) {
56              left = (*p).left();
57            }
58            else {
59              owner = (*p).owner();
60            }
61            if((auto_splice_out && do_splice()) || forced) {
62              (*p).parent() = 0;
63              (*p).left() = 0;
64            }
65          }
66          hole_type(): da(0), parent(0), current(0), left(0) {}
67          bool is_root() const { return parent == 0; }
68          bool do_splice() { return !is_root(); }
69        };
70        hole_type splice_out(bool forced = false) {
71          // Note: This function leaves the underlying tree broken,
72          // untraversable, and unprintable until splice_in.
73          assert(this != 0);
74          hole_type hole(this,true,forced);
75          return hole;
76        }
77        template <typename H>
78        void splice_in(hole_type& hole, H&, bool forced = false) {
79          if(!(hole.do_splice() || forced)) return;
80          assert(hole.current != 0);
81          (*this).parent() = hole.parent;
82          if (hole.parent != 0 && (*hole.parent).left() == hole.current) {
83            (*hole.parent).left() = this;
84          }
85          if (hole.parent != 0 && (*hole.parent).right() == hole.current) {
86            (*hole.parent).right() = this;
87          }
88          if (hole.parent == 0) {
89            (*this).owner() = hole.owner;
90          }
91          else {
92            (*this).left() = hole.left;
93            if (hole.left != 0) {
94              (*hole.left).parent() = this;
95            }
96          }
97        }
98
99      protected:
100
101       N* left() const { return left_; }
102       N*& left() { return left_; }
103       N* right() const { return right_; }
104       N*& right() { return right_; }
105       N* parent() const { return parent_; }
106       N*& parent() { return parent_; }
107
108     public:
109
110       pwh_node_base() : left_(0), parent_(0), right_(0) {}
111
112       static size_type footprint() { return sizeof(N); }
113       bool is_root() const { return parent_ == 0; }
114       bool is_leaf() const { return right_ == 0; }
115
```

```
116         static bool is_spooky(N const* p) { return 0<(long long)p && (long long)p <1000; }
117
118         bool is_valid() const {
119           bool node_is_valid = this != 0
120             && !(is_spooky(this) || is_spooky(parent_)
121             || is_spooky(left_) || is_spooky(left_));
122           assert(node_is_valid);
123           return(node_is_valid);
124         }
125
126         bool tree_valid() const {
127           bool tree_node_valid(true);
128           if ((*this).parent_ != 0) {
129             tree_node_valid &= (*(*this).parent_).left_ == this ||
130               (*(*this).parent()).right() == this;
131           }
132           if ((*this).left_ != 0) {
133             tree_node_valid &= (*(*this).left_).parent_ == this;
134           }
135           if ((*this).right_ != 0) {
136             tree_node_valid &= (*(*this).right_).parent_ == this;
137           }
138           assert(tree_node_valid);
139           return tree_node_valid;
140         }
141
142         void* owner() const {
143           assert(is_valid()); assert(!careful || is_root());
144           return owner_;
145         }
146         void*& owner() {
147           assert(is_valid()); assert(!careful || is_root());
148           return owner_;
149         }
150
151         N* join(N* weaker) {
152           assert(is_valid()); assert(!careful || is_root());
153           assert((*weaker).is_valid()); assert(!careful || (*weaker).is_root());
154           N* p(this);
155           N* c((*p).right());
156           if (c != 0) { (*c).parent() = weaker; }
157           (*weaker).left() = c;
158           (*p).right() = weaker;
159           (*weaker).parent() = p;
160           return this;
161         }
162
163         N* most_significant_child() const {
164           // Desc: locate weaker subtree before an actual split.
165           assert(is_valid());
166           return (*this).right();
167         }
168
169         N* split() {
170           assert(is_valid()); assert(!careful || is_root());
171           assert((*this).right() != 0); //??
172           N* p(this);
173           N* q((*p).right());
174           // wrong layer for doing this: mark_store.unmark(q);
175           N* r((*q).left());
176           (*p).right() = r;
177           if (r != 0) { (*r).parent() = p; }
178           (*q).parent() = 0;
179           (*q).left() = 0;
180           return q;
181         }
182
183         N* swap_roots(N* q) {
184           N* p = this;
185           assert((*p).is_root());
186           assert((*q).is_root());
187           N* c = (*p).right();
188           N* g = (*q).right();
189           (*p).right() = g;
```

```
190           (*q).right() = c;
191           if (c != 0) { (*c).parent() = q; }
192           if (g != 0) { (*g).parent() = p; }
193           return this;
194         }
195
196         template <class Boolean>
197         N* construct(size_type& n, const Boolean calc_size) {
198           // Desc: Vuillemin "construct". T(n): O(lg n). ES: no throw.
199           //       Isolate the root from its childs.
200           // Ret:  A root list in the reverse order of an recursive split
201           //       + total number of nodes including the root.
202           assert(is_valid());
203           N* r(0);
204           if(calc_size) n = 1;
205           for(N* c(right_); c != 0; ) {
206             N* d((*c).left_);
207             (*c).parent_ = 0;
208             (*c).left_ = r;
209             r = c;
210             if(calc_size) n <<= 1;
211             c = d;
212           }
213           right_ = 0;
214           return r;
215         }
216
217         N* release_root() {
218           // Desc: Vuillemin construct with reverse root list ordering.
219           N* p(this);
220           N* q((*p).right_);
221           (*p).right_ = 0;
222           (*q).parent_ = 0;
223           return q;
224         }
225
226         N* release_subheap() {
227           // Desc: disconnect reverse root list tail.
228           N* p(this);
229           N* q((*p).left_);
230           (*p).left_ = 0;
231           if (q != 0) (*q).parent_ = 0;
232           return q;
233         }
234
235         N* distinguished_descendant() const {
236           // Note: this is part of something in pqfw-extract(p)
237           //       that perhaps might be done more efficiently using the
238           //       Vuillemin "construct".
239           assert(is_valid());
240           N const* q = this;
241           assert(q != 0);
242           if ((*q).is_root()) {
243             return (*q).right();
244           }
245           return (*q).left();
246         }
247
248         N* distinguished_ancestor() const {
249           assert(is_valid());
250           N const* q = this;
251           assert(q != 0);
252           N* p = (*q).parent();
253           while (p != 0 && (*p).left() == q) {
254             q = p;
255             p = (*p).parent();
256           }
257           return p;
258         }
259
260         N* promote_alt(N* p) { // Vuillemin compatible.
261           promote(p);
262           return (*this).distinguished_ancestor();
263         }
```

```
264
265         N* promote(N* p) { // CPH STL compatible.
266           // Desc: this is the Jyrki version of promote which is faster than my version of
                this operation.
267           N* q = this;
268           assert(p == (*q).distinguished_ancestor());
269           if (p == (*q).parent()) {
270             assert((*p).right() == q);
271             N* a = (*p).parent();
272             N* b = (*p).left();
273             N* f = (*q).left();
274             N* g = (*q).right();
275             (*p).parent() = q;
276             (*p).left() = f;
277             (*p).right() = g;
278             (*q).parent() = a;
279             (*q).left() = b;
280             (*q).right() = p;
281             if (a != 0) {
282               if ((*a).right() == p) {
283                 (*a).right() = q;
284               }
285               else {
286                 if (! (*p).is_root()) {
287                   (*a).left() = q;
288                 }
289               }
290             }
291             if (b != 0 && (! (*q).is_root())) {
292               (*b).parent() = q;
293             }
294             if (f != 0) {
295               (*f).parent() = p;
296             }
297             if (g != 0) {
298               (*g).parent() = p;
299             }
300           }
301           else {
302             N* a = (*p).parent();
303             N* b = (*p).left();
304             N* c = (*p).right();
305             N* e = (*q).parent();
306             N* f = (*q).left();
307             N* g = (*q).right();
308             (*p).parent() = e;
309             (*p).left() = f;
310             (*p).right() = g;
311             (*q).parent() = a;
312             (*q).left() = b;
313             (*q).right() = c;
314             if (a != 0) {
315               if ((*a).right() == p) {
316                 (*a).right() = q;
317               }
318               else {
319                 if (! (*p).is_root()) {
320                   (*a).left() = q;
321                 }
322               }
323             }
324             if (b != 0 && (! (*q).is_root())) {
325               (*b).parent() = q;
326             }
327             if (c != 0) {
328               (*c).parent() = q;
329             }
330             if (e != 0 && (*e).left() == q) {
331               (*e).left() = p;
332             }
333             if (e != 0 && (*e).right() == q) {
334               (*e).right() = p;
335             }
336             if (f != 0) {
```

```
337             (*f).parent_ = p;
338           }
339           if (g != 0) {
340             (*g).parent_ = p;
341           }
342         }
343         return q;
344       }
345
346       N* promote_first(N* d) {
347         // Desc: first Asger version, note: Jyrkis promote is better.
348         /*
349         * Desc: exchange position with first distinguished ancestor.
350         * T(n): O(1)? ES: no throw.
351         *
352         *  d != p:        (g)                    d == p:       o
353         *                  \                                   \
354         *           ------(d)----                          -----(g)-----
355         *          /       .    \                        /             \
356         *       --o--      .     --o--                 -(d)-          --o--
357         *      /     \     .    /     \               /     \        /     \
358         *     o       o    . (p)      o             o       (n)     o       o
359         *    / \     / \   . / \     / \           / \     / \     / \     / \
360         *   o   o   o   o  (n)  o   o   o         o   o   o   o   o   o   o   o
361         */
362         assert(this != 0);
363         assert(d != 0);
364         assert(d == distinguished_ancestor()); // T = lg?
365         N* g = (*d).parent_;
366         if(d == (*this).parent_) {
367           // connect to parent of distinguished parent.
368           (*this).parent_ = g;
369           if(g != 0) {
370             if(d == (*g).right_) std::swap((*g).right_, (*d).right_);
371             else std::swap((*g).left_, (*d).right_);
372           }
373           // transfer right part.
374           (*d).right_ = (*this).right_;
375           if((*d).right_ != 0) (*(*d).right_).parent_ = d;
376           (*this).right_ = d;
377           (*d).parent_ = this;
378         } else {
379           // swap parents with distinguished parent.
380           if(g != 0) {
381             if(d == (*g).right_) std::swap((*g).right_, (*(*this).parent_).left_);
382             else std::swap((*g).left_, (*(*this).parent_).left_);
383           } else (*(*this).parent_).left_ = d;
384           std::swap((*this).parent_, (*d).parent_);
385           // swap right parts.
386           std::swap((*this).right_, (*d).right_);
387           if((*d).right_ != 0) std::swap((*(*this).right_).parent_, (*(*d).right_).parent_);
388           else (*(*this).right_).parent_ = this;
389         }
390         // swap left parts.
391         std::swap((*this).left_, (*d).left_);
392         if(g != 0) {
393           if((*d).left_ != 0 && (*this).left_ != 0)
394             std::swap((*(*this).left_).parent_, (*(*d).left_).parent_);
395           else if((*this).left_ != 0) (*(*this).left_).parent_ = this;
396           else if((*d).left_ != 0) (*(*d).left_).parent_ = d;
397         } else {
398           if((*d).left_ != 0) (*(*d).left_).parent_ = d;
399         }
400
401         return this;
402       };
403
404       N /*const*/ * root() /*const*/ {
405         assert(is_valid());
406         N /*const*/* p = this;
407         assert(p != 0);
408         while (! (*p).is_root()) {
409           p = (*p).parent();
410         }
```

```
411           return p;
412         }
413         N const* root() const {
414           assert(is_valid());
415           N const* p = this;
416           assert(p != 0);
417           while (! (*p).is_root()) {
418             p = (*p).parent();
419           }
420           return p;
421         }
422
423
424
425         N* successor() const {
426           // Desc: "right child first"-ordered traversal.
427           // T(n): O(1). ES: no throw.
428           assert(is_valid());
429           if(most_significant_child() != 0)
430             return most_significant_child();
431           else {
432             N* d = distinguished_ancestor();
433             if(d != 0 && !(*d).is_root())
434               return (*d).distinguished_descendant();
435           }
436           return 0;
437         }
438
439         height_type height() const {
440           // Desc: Slow height, usefull for assertions. ES: no throw.
441           assert(is_valid());
442           height_type h(0);
443           for(N* c(right_); c != 0; c = (*c).right_) { ++h; }
444           return h;
445         }
446
447
448         std::string str() const { return (*this).str<N>(); }
449
450       protected:
451
452         template <typename D>
453         std::string str() const {
454           std::ostringstream os;
455           if(this == 0) { os << "nil"; }
456           else {
457             if((parent_ != 0 && left_ != 0) || right_ != 0) {
458               os << "<";
459               if(parent_ != 0 && left_ != 0) os << (*static_cast<D *const>(left_)).str();
460               os << ",";
461               if(right_ != 0) os << "␣" << (*static_cast<D *const>(right_)).str();
462               os << ">";
463             }
464           }
465           return os.str();
466         }
467       };
468
469       template <>
470       struct node_traits<pwh_node_base>  { enum { has_splice = 1 }; enum { has_owner = 1 };
          enum { is_root_listable = 1 }; };
471     }
472   }
473   #endif
```

A.28 redundant_binary_number_system.hpp

```
1   #ifndef _CPHSTL_REDUNDANT_BINARY_NUMBER_SYSTEM_H_
2   #define _CPHSTL_REDUNDANT_BINARY_NUMBER_SYSTEM_H_
3
4   /*
5   Desc: Vuillemin style redundant binary number system
6         with carry bits handled in constant time
```

```
7         (or what ever the join schedule policy gives).
8   Auth: Asger Bruun 2009-2010
9
10  Ref:  Amr Elmasry and Claus Jensen and Jyrki Katajainen,
11        Relaxed weak queues: {A}n alternative to run-relaxed heaps (2005).
12  */
13
14  #include <array>
15
16  #include "root_list.hpp" // root_list
17  #include "binary_number_system.hpp" // for cross over
18  #include "join_schedule_policies.hpp"
19
20  namespace cphstl {
21
22
23    template<typename Z = std::size_t>
24    struct redundant_binary_number_base {
25      Z accu, carry;
26
27      redundant_binary_number_base() : accu(0), carry(0) {}
28      redundant_binary_number_base(Z size) : accu(size), carry(0) {}
29      redundant_binary_number_base(redundant_binary_number_base const& n)
30        : accu(n.accu), carry(n.carry) {}
31      redundant_binary_number_base(redundant_binary_number_base const& n, int offset)
32        : accu(n.accu >> offset), carry(n.carry >> offset) {}
33
34      Z value() const { return (*this).accu+(*this).carry; }
35
36      void clear() { accu = 0; carry = 0; }
37
38      void increment() {
39        assert(is_valid());
40        if(((*this).accu & 1) == 0) ++(*this).accu;
41        else ++(*this).carry;
42      }
43
44      void decrement() {
45        assert(is_non_zero());
46        Z n(1 << trailing_zeros(accu | carry)); // first bit value
47        if(n & carry) { carry -= n; accu += n; };
48        --accu;
49      }
50
51      void subtract(Z bit_value) {
52        assert(pop_cnt(bit_value) == 1);
53        assert((((*this).accu | (*this).carry) & bit_value) != 0);
54        if(bit_value & carry) carry -= bit_value;
55        else accu -= bit_value;
56      }
57
58      bool is_valid() const {
59        assert(((*this).accu & (*this).carry) == (*this).carry);
60        assert((*this).accu >= (*this).carry);
61        return (((*this).accu & (*this).carry) == (*this).carry)
62          && ((*this).accu >= (*this).carry);
63      }
64      bool is_non_zero() const { return (accu | carry) != 0; }
65
66      void swap(redundant_binary_number_base & n2) {
67        std::swap(accu, n2.accu); std::swap(carry, n2.carry);
68      }
69
70      void propagate(Z carry_bit_value) {
71        assert(pop_cnt(carry_bit_value) == 1);
72        assert((((*this).accu & (*this).carry) & carry_bit_value) != 0);
73        (*this).accu -= carry_bit_value;
74        (*this).carry -= carry_bit_value;
75        carry_bit_value <<= 1;
76        assert((((*this).accu & (*this).carry) & carry_bit_value) == 0);
77        if(((*this).accu & carry_bit_value) == 0) (*this).accu |= carry_bit_value;
78        else (*this).carry |= carry_bit_value;
79      }
80
```

```
81      class bit_scanner : protected redundant_binary_number_base /*<typename Z>*/ {
82      public:
83
84        typedef typename size_traits<sizeof(Z)*8>::height_type bit_pos_type;
85        static const bit_pos_type none = ~ bit_pos_type(0);
86
87      private:
88
89        bit_pos_type bit_pos_;
90        bool is_carry_;
91
92      public:
93
94        bit_scanner(redundant_binary_number_base const& d, bit_pos_type offset)
95          : redundant_binary_number_base(d, offset)
96        {
97          if((accu | carry) == 0) { bit_pos_ = none; is_carry_ = false; }
98          else {
99            bit_pos_ = offset;
100           is_carry_ = (((*this).accu & (*this).carry) % 2 == 1);
101           next_bit();
102         }
103       }
104
105       Z current() { return bit_pos_ == none?0:1 << bit_pos_; }
106       bit_pos_type bit_pos() { return bit_pos_; }
107
108       bool is_carry() { return is_carry_; }
109       bool has_more_carry() { return (*this).carry != 0; }
110
111       bool is_next_bit_pos_non_zero() {
112         return (((*this).accu | (*this).carry) & 2) != 0;
113       }
114
115       void next_bit() {
116         assert(bit_pos_ != none);
117
118         if(((*this).accu | (*this).carry) == 0) {
119           bit_pos_ = none; return;
120         } else if((((*this).accu | (*this).carry) & 1) == 0) {
121           bit_pos_type delta = (bit_pos_type) trailing_zeros(accu | carry);
122           (*this).accu >>= delta; (*this).carry >>= delta;
123           bit_pos_+= delta;
124           is_carry_ = (((*this).accu & (*this).carry) % 2 == 1);
125         }
126         if((*this).carry % 2) { --(*this).carry; }
127         else { assert(((*this).accu % 2) == 1); --(*this).accu; }
128       }
129
130     private:
131
132       friend std::ostream & operator <<
133         (std::ostream & os, bit_scanner const& bs)
134       {
135         bit_scanner tmp(bs);
136         while(tmp.current()) {
137           os << tmp.current() << "␣"; tmp.next_bit();
138         }
139         return os;
140       }
141     };
142
143   private:
144
145     friend std::ostream & operator <<
146       (std::ostream & os, redundant_binary_number_base const& d)
147     {
148       os << d.accu << "+" << d.carry << "␣=␣" << d.value();
149       return os;
150     }
151   };
152
153   ////////////////////////////////////////////////
154
```

```
155     template<
156       template <typename E2> class JS = join_schedule_fat
157     >
158     struct js_policy {
159       template<typename C, typename E>
160       // where E and C have the same type of comparator and value.
161       class redundant_binary_number : public E::root_list_type {
162         redundant_binary_number(redundant_binary_number const&);
163         redundant_binary_number& operator = (const redundant_binary_number&);
164
165       public:
166
167         enum { number_system = 3 };
168
169         typedef typename E::comparator_type comparator_type;
170         typedef typename E::root_list_type root_list_type;
171         typedef typename root_list_type::back_inserter back_inserter;
172         typedef typename E::size_type size_type;
173         typedef typename E::pointer pointer;
174
175         typedef redundant_binary_number_base<size_type> number_base_type;
176         typedef typename number_base_type::bit_scanner bit_scanner;
177
178         //slower is: typedef redundant_binary_number<C,E> fast_number_type;
179         typedef binary_number<C,E> fast_number_type;
180         typedef typename fast_number_type::bit_scanner fast_bit_scanner;
181
182         redundant_binary_number(pointer list, size_type accu, size_type carry)
183           : root_list_type(list), size_(accu,carry), js() { assert(is_valid()); }
184
185         redundant_binary_number(pointer list, number_base_type const& size)
186           : root_list_type(list), size_(size), js() { assert(is_valid()); }
187
188         bool is_valid() /*const*/ {
189           bool valid = (*this).size_.is_valid();
190           assert(valid);
191           bit_scanner bs((*this).size_, 0);
192           valid &= root_list_type::is_valid_bs(bs);
193           assert(valid);
194           valid &= (*this).js.is_valid(this);
195           return valid;
196         }
197
198         size_type size() const { return (*this).size_.value(); }
199         size_type max_size() const {
200           // Note: It is clear to the LWG that the value returned by max_size()
201           // :      can't change from call to call.
202           // Ref:  www.open-std.org/jtc1/sc22/wg21/docs/lwg-closed.html#197
203           return std::numeric_limits<size_type>::max() / E::footprint();
204         }
205
206         void swap(redundant_binary_number<C,E> & h2) {
207           assert((*this).is_valid()); assert(h2.is_valid());
208           root_list_type::swap(h2);
209           (*this).js.swap(h2.js);
210           (*this).size_.swap(h2.size_);
211         }
212
213       protected:
214
215         number_base_type size_;
216         static comparator_type comparator;
217
218         typedef JS<E> join_schedule_type;
219
220         //friend class join_schedule_type; // this isnt valid gcc.
221
222         friend class join_schedule_ultralight<E>;
223         friend class join_schedule_light<E>;
224         friend class join_schedule_medium<E>;
225         friend class join_schedule_fat<E>;
226
227         join_schedule_type js;
228
```

```
229     private:
230
231       // adder
232       typedef typename root_list_type::adder_access_type adder_access_type;
233       typedef typename root_list_type::adder_access_ref adder_access_ref;
234       static void accept_carry(back_inserter & nxt, pointer& c) {
235         nxt.add(c); c = 0;
236       }
237       static void accept_res(back_inserter & nxt, adder_access_ref l) {
238         nxt.add(l.skip());
239       }
240       static void prop_carry(pointer& c, adder_access_ref l) {
241         c = (*l.eject()).join(c, comparator);
242       }
243       static void prop_carry2(pointer& c, adder_access_ref l) {
244         c = (*c).join(l.eject(), comparator);
245       }
246       static void construct_carry(pointer& c, adder_access_ref l1, adder_access_ref l2) {
247         c = (*l1.eject()).join(l2.eject(), comparator);
248       }
249       static void join_carry(pointer& c1, pointer& c2) {
250         c1 = (*c1).join(c2, comparator);
251         c2 = 0;
252       }
253
254     protected:
255
256       void add(binary_number<C,E>& b) {
257         // T(n,m) = O(max(lg m, lg n)).
258         // Desc: add with faster binary number type (almost obsolete).
259
260         assert((*this).is_valid()); assert(b.is_valid());
261
262         assert(b.accu != 0);
263         int bm(bit_scan_reverse(b.accu));
264         while(((*this).size_.carry != 0) && (trailing_zeros((*this).size_.carry)<bm))
265           js.propagate_carry(this); // !: unfortunate hack
266
267
268         adder_access_type r1(*this), r2(b);
269         number_base_type n1((*this).size_);
270         size_type n2(b.accu);
271         pointer c(0);
272         root_list_type res(0);
273         back_inserter nxt(res);
274         while((n1.accu != 0 && n2 != 0) || (c != 0)) {
275           if(n1.carry % 2 != 0) accept_res(nxt, r1);
276           switch((c == 0?0:4) + ((n2 % 2) << 1) + (n1.accu % 2)) {
277           case 0 /*000*/: assert(n1.carry % 2 == 0); break;
278           case 1 /*001*/: accept_res(nxt, r1); break;
279           case 2 /*010*/: accept_res(nxt, r2); break;
280           case 3 /*011*/: construct_carry(c, r1, r2); break;
281           case 4 /*100*/: accept_carry(nxt, c); break;
282           case 5 /*101*/: prop_carry(c, r1); break;
283           case 6 /*110*/: prop_carry2(c, r2); break;
284           case 7 /*111*/: accept_res(nxt, r1); prop_carry(c, r2); break;
285           default: assert(0); break;
286           }
287           n1.accu >>= 1; n1.carry >>= 1; n2 >>= 1;
288         }
289         if(n1.accu != 0) {
290           assert(n2 == 0); assert(r2.empty());
291           accept_res(nxt, r1);
292           (*this).clear();
293         } else if(n2 != 0) {
294           assert(r1.empty());
295           accept_res(nxt, r2);
296           b.clear();
297         } else {
298           assert(r1.empty());
299           assert(r2.empty());
300         }
301         (*this).size_.accu += b.accu;
302         b.accu = 0;
```

```
303           (*this).root_list_type::swap(res);
304
305           // transfer outdated carry to accu.
306           size_type a((*this).size_.accu);
307           (*this).size_.accu |= (*this).size_.carry;
308           (*this).size_.carry &= a;
309           if((*this).size_.accu != a) js.scan_drop(this);
310
311           assert((*this).is_valid()); assert(b.is_valid());
312         }
313
314         void add_inefficient(redundant_binary_number& b) {
315           // T(n,m) = O(min(lg m, lg n)).
316           // Desc: Vuillemin adder, but with a design flaw in the use
317           //       of the number system, and with a costly fix in extra compares.
318
319           assert((*this).is_valid()); assert(b.is_valid());
320
321           if(b.size_.carry > (*this).size_.accu)
322             (*this).swap(b); // pick join schedule with highest digit.
323
324           if(b.size_.accu == 0) return; // copy construction from mpq
325           int bm(bit_scan_reverse(b.size_.accu));
326           while(((*this).size_.carry != 0) && (trailing_zeros((*this).size_.carry)<bm))
327             js.propagate_carry(this); // !: unfortunate hack
328
329
330           number_base_type n1((*this).size_), n2(b.size_);
331           adder_access_type r1(*this), r2(b);
332           //root_list_type & r1(*this), & r2(b);
333           pointer c1(0), c2(0);
334           root_list_type res(0);
335           back_inserter nxt(res);
336           while(((n1.accu | n1.carry) != 0
337             && (n2.accu | n2.carry) != 0) || c1 != 0 || c2 != 0)
338           {
339             // if(c1 != 0 && c2 != 0) __break_here;
340             if(n1.carry % 2 != 0) accept_res(nxt, r1);
341             switch((c2 == 0?0:16) + (c1 == 0?0:8) + ((n2.carry % 2) << 2)
342               + ((n2.accu % 2) << 1) + (n1.accu % 2)) {
343         case 0  /*00000*/: assert(n1.carry % 2 == 0); break;
344         case 1  /*00001*/: accept_res(nxt, r1); break;
345         case 2  /*00010*/: accept_res(nxt, r2); break;
346         case 3  /*00011*/: construct_carry(c1, r1, r2); break;
347         case 4  /*00100*/: accept_res(nxt, r2); break;
348         case 5  /*00101*/: construct_carry(c1, r1, r2); break;
349         case 6  /*00110*/: construct_carry(c1, r2, r2); break;
350         case 7  /*00111*/: accept_res(nxt, r2); construct_carry(c1, r1, r2); break;
351         case 8  /*01000*/: accept_carry(nxt, c1); break;
352         case 9  /*01001*/: prop_carry(c1, r1); break;
353         case 10 /*01010*/: prop_carry(c1, r2); break;
354         case 11 /*01011*/: accept_res(nxt, r1); prop_carry(c1, r2); break;
355         case 12 /*01100*/: prop_carry(c1, r2); break;
356         case 13 /*01101*/: accept_res(nxt, r1); prop_carry(c1, r2); break;
357         case 14 /*01110*/: accept_res(nxt, r2); prop_carry(c1, r2); break;
358         case 15 /*01111*/: prop_carry(c1, r2); construct_carry(c2, r1, r2); break;
359         case 16 /*10000*/: accept_carry(nxt, c2); break;
360         case 17 /*10001*/: prop_carry(c2, r1); break;
361         case 18 /*10010*/: prop_carry(c2, r2); break;
362         case 19 /*10011*/: accept_res(nxt, r1); prop_carry(c2, r2); break;
363         case 20 /*10100*/: prop_carry(c2, r2); break;
364         case 21 /*10101*/: accept_res(nxt, r1); prop_carry(c2, r2); break;
365         case 22 /*10110*/: accept_res(nxt, r2); prop_carry(c2, r2); break;
366         case 23 /*10111*/: accept_res(nxt, r1); prop_carry(c2, r2); prop_carry(c2, r2); break;
367         case 24 /*11000*/: join_carry(c1, c2); break;
368         case 25 /*11001*/: join_carry(c1, c2); accept_res(nxt, r1); break;
369         case 26 /*11010*/: join_carry(c1, c2); accept_res(nxt, r2); break;
370         case 27 /*11011*/: join_carry(c1, c2); construct_carry(c2, r1, r2); break;
371         case 28 /*11100*/: join_carry(c1, c2);  accept_res(nxt, r2); break;
372         case 29 /*11101*/: join_carry(c1, c2); construct_carry(c2, r1, r2); break;
373         case 30 /*11110*/: join_carry(c1, c2); construct_carry(c2, r2, r2); break;
374         // case 31 /*11111*/: assert(0); break; // impossible
375         default: assert(0); break; // impossible
376             }
```

```
377             n1.accu >>= 1; n1.carry >>= 1;
378             n2.accu >>= 1; n2.carry >>= 1;
379           }
380           if(n1.accu != 0) {
381             assert(n2.value() == 0); assert(r2.empty());
382             accept_res(nxt, r1);
383             (*this).clear();
384           } else if((n2.accu | n2.carry) != 0) {
385             assert(n2.carry == 0); // or rebuild rest of join schedule.
386             assert(r1.empty());
387             accept_res(nxt, r2);
388             b.clear();
389           } else {
390             assert(r1.empty());
391             assert(r2.empty());
392           }
393           (*this).size_.accu += b.size_.value();
394           b.size_.clear();
395           b.js.clear();
396           root_list_type::swap(res);
397
398           // transfer outdated carry to accu.
399           size_type a((*this).size_.accu);
400           (*this).size_.accu |= (*this).size_.carry;
401           (*this).size_.carry &= a;
402           if((*this).size_.accu != a) js.scan_drop(this);
403
404           assert((*this).is_valid()); assert(b.is_valid());
405         }
406
407         struct forward_adder_stack_entry {
408           pointer carry_node;
409           int carry_pos;
410         };
411
412         void add(redundant_binary_number& b) {
413           // Desc: New forward adder.
414           // T(n,m) = O(min(lg m, lg n)).
415           if(b.size_.accu > (*this).size_.accu) (*this).swap(b);
416           if(b.size_.accu == 0) return;
417
418           assertion_help(size_type expected((*this).size()+b.size()));
419
420           number_base_type n1((*this).size_), n2(b.size_);
421           adder_access_type l1(*this), l2(b);
422
423           root_list_type res(0);
424           back_inserter nxt(res);
425
426           int bit_count(0), zero_written(0), bit_pos(0);
427           pointer bits[6];
428
429           const int max_stack = (sizeof(size_type)*CHAR_BIT/2) - (static_logb<sizeof(E)>::
                 value/2);
430           // unfortunately not possible: "std::stack<forward_adder_stack_entry, std::array<
                 forward_adder_stack_entry,max_stack> > stk;"
431           //forward_adder_stack_entry stk[max_stack];
432           std::array<forward_adder_stack_entry,max_stack> stk;
433           int top(0);
434
435           (*this).size_.clear();
436
437           while(/*n1.accu |*/ n2.accu | bit_count) {
438             if(n1.accu & 1) {
439               bits[bit_count++] = l1.eject();
440               if(n1.carry & 1) {
441                 bits[bit_count++] = l1.eject();
442               }
443             }
444             if(n2.accu & 1) {
445               bits[bit_count++] = l2.eject();
446               if(n2.carry & 1) {
447                 bits[bit_count++] = l2.eject();
448               }
```

```
449             }
450             if((bit_count & 1) != 0) { // odd
451               nxt.add(bits[--bit_count]);
452               (*this).size_.accu |= (1 << bit_pos);
453             } else { // even
454               if(bit_count == 0) {
455                 zero_written = 1;
456               } else if(zero_written) {
457                 nxt.add(bits[--bit_count]); nxt.add(bits[--bit_count]);
458                 stk[top].carry_node = bits[bit_count];
459                 stk[top].carry_pos = bit_pos;
460                 ++top;
461                 (*this).size_.carry |= (1 << bit_pos);
462                 (*this).size_.accu |= (1 << bit_pos);
463                 zero_written = 0;
464               }
465             }
466             bit_count >>= 1;
467             switch(bit_count) {
468             case 0: break;
469             case 1: bits[0] = (*bits[0]).join(bits[1], comparator); break;
470             case 2: bits[0] = (*bits[0]).join(bits[1], comparator);
471                     bits[1] = (*bits[2]).join(bits[3], comparator); break;
472             default: assert(0); break;
473             }
474             n1.accu >>= 1; n1.carry >>= 1;
475             n2.accu >>= 1; n2.carry >>= 1;
476             ++bit_pos;
477           }
478           if(n1.accu) { // accept rest of root list as is.
479             pointer p(l1.skip());
480             nxt.add(p);
481             while(n1.carry) { // save rest of join schedule.
482               if(n1.carry & 1) {
483                 stk[top].carry_node = p;
484                 stk[top].carry_pos = bit_pos;
485                 ++top;
486                 p = l1.skip(); p = l1.skip();
487               } else if(n1.accu & 1) {
488                 p = l1.skip();
489               }
490               n1.accu >>= 1; n1.carry >>= 1;
491               ++bit_pos;
492             }
493           }
494           b.size_.clear();
495           b.js.clear();
496           (*this).js.clear();
497           (*this).root_list_type::swap(res);
498           while(top != 0) { // rebuild join schedule
499             --top;
500             (*this).js.push(stk[top].carry_pos, stk[top].carry_node);
501           }
502
503           assert((*this).size()==expected);
504         }
505
506         size_type remove_root(pointer r) {
507           // Desc: Remove root from root list and return its size.
508           //       This is used from Vuillemin extract.
509           //       Locating corresponding bit requires a list scan,
510           //       therefore constant time is impossible.
511           assert((*this).is_valid()); assert((*r).is_root());
512           pointer y((*this).begin()), z((*this).end());
513           bit_scanner bs((*this).size_, 0);
514           while(y != r) {
515             bs.next_bit(); z = y; y = next_get(y);
516           }
517           size_type delta(bs.current());
518           bool was_carry(((*this).size_.carry & delta) != 0);
519           (*this).size_.subtract(delta);
520           if(was_carry) {
521             assert(bs.is_carry());
522             js.drop(this, bs.bit_pos());
```

```
523           }
524           root_list_type::remove_after(z);
525           assert((*this).is_valid());
526           return delta;
527         }
528
529         void promoted_to_root(pointer replacement, pointer d) {
530           js.replace(replacement,d);
531           root_list_type::promoted_to_root(replacement,d);
532           assert((*this).is_valid());
533         }
534
535         void increment(pointer p) {
536           // Opti: insert does never meld.
537           assert((*this).is_valid()); assert(p != 0);
538           assert((*p).is_root()); assert((*p).height() == 0);
539
540           root_list_type::inject(p);
541           (*this).size_.increment();
542           if(((*this).size_.accu & (*this).size_.carry) % 2 == 1) {
543             js.push(0,p);
544           }
545           js.propagate_carry(this);
546
547           assert((*this).is_valid());
548         }
549
550         pointer decrement() {
551           // Opti: Vuillemin meld is never needed because the tree
552           //       extracted is always the least significant digit.
553           assert((*this).is_valid());
554           assert(!(*this).empty());
555
556           pointer q(E::root_list_type::begin());
557           (*this).size_.decrement();
558           if (q == js.top_node()) js.pop(this);
559
560           pointer p(E::root_list_type::eject());
561           assert(p == q);
562
563           pointer t((*p).most_significant_child());
564           size_type dummy(1);
565           if(t != 0) { // ie. !(*p).is_leaf()
566             pointer r((*p).construct(dummy, constant<bool,false>()));
567             inject_range(r,t);
568           }
569
570           assert(is_valid()); assert((*p).is_root()); assert((*p).height() == 0);
571           return p;
572         }
573
574       private:
575
576         /*
577         // dont offer eject unless the cph stl extract needs it:
578         pointer eject() {
579           pointer p(root_list_type::eject());
580           if (p == js.top()) js.pop();
581           return p;
582         }
583         */
584
585         friend std::ostream & operator <<
586           (std::ostream & os, redundant_binary_number const& hs)
587         {
588           os << "size(" << hs.size_ << ")␣";
589           os << "length(" << hs.length() << ")␣";
590           bit_scanner bs(hs.size_, 0);
591           pointer r(hs.begin());
592           while(r != 0) {
593             os << "[" << size_type(bs.bit_pos()) << ":" << (*r) << "]␣";
594             r = (*r).next_root_get();
595             bs.next_bit();
596           }
```

```
597           os << "\n";
598           return os;
599         }
600       }; // redundant_binary_number
601     }; // js_policy
602
603     template<template <typename E2> class JS>
604     template<typename C, typename E>
605     typename cphstl::js_policy<JS>::template redundant_binary_number<C,E>::comparator_type
606       js_policy<JS>::redundant_binary_number<C,E>::comparator = C();
607
608   } // namespace cphstl
609   #endif
```

A.29 root_list.hpp

```
1   #ifndef __CPHSTL_ROOT_LIST_H__
2   #define __CPHSTL_ROOT_LIST_H__
3
4   /*
5   Desc: Vuillemin style root list.
6   Auth: Asger Bruun 2009-2010
7   Ref:  Jean Vuillemin. A data structure for manipulating priority queues.
8         Communications of the ACM 21 (1978), 309-315.
9   Note: This is the unidirectional version.
10        The bidirectional version has temporarely branched to
11        "root_list_bidir.hpp" because serious trouble combining the
12        two versions in one source.
13  */
14
15  namespace cphstl {
16
17    template <class T, T V>
18    struct constant {
19        operator T() const { return V; }
20    };
21
22    namespace pqfw_node {
23
24      template <typename D>
25      class root_list_node
26      {
27        typedef root_list_node N;
28
29      // protected:
30      public: // !!: temporarely public
31
32        union {
33          D* next_;
34          void* owner_;
35        };
36
37      public:
38
39        static bool is_spooky(void const* p) {
40          return 0<(long long)p && (long long)p <1000;
41        }
42
43        static bool is_valid(N const* const p) {
44          bool root_list_node_valid = p != 0
45            && !(is_spooky(p) || is_spooky((*p).next_));
46          assert(root_list_node_valid);
47          return(root_list_node_valid);
48        }
49        bool is_valid() const {
50          return is_valid(this);
51        }
52
53        typedef root_list_node root_list_base;
54
55        template<typename E2>
56        struct rebind_encapsulator_type {
57          typedef root_list_node<E2> other;
```

```
58        };
59
60
61        root_list_node() : next_(0) {}
62        root_list_node(D* next) : next_(next) {}
63
64        D*& next() {
65          assert((*this).is_valid());
66          return (*this).next_;
67        }
68        D* next() const {
69          assert((*this).is_valid());
70          return (*this).next_;
71        }
72        D* next_root_get() {
73          assert((*this).is_valid());
74          return (*this).next_;
75        }
76        void next_root_set(D* r) {
77          assert((*this).is_valid());
78          (*this).next_ = r;
79        }
80      };
81
82      template <class T> struct node_traits {
83        // special pqfw features expensive to implement.
84        enum { has_splice = 0 }; enum { has_owner = 0 }; enum { is_root_listable = 0 };
85      };
86    } // namespace pqfw_node
87
88    using namespace pqfw_node;
89
90    template<typename E, bool bidir>
91    class root_list {
92
93    private:
94
95      E* lst_;
96
97      E* before_head() { // root list behaving as a list node.
98        assertion_help(E* r((E*)&lst_);)
99        assert((void*)r == (void*)&((*r).owner())); // require next as first field (better if
              the test was static).
100       return (E*)&lst_;
101     }
102     E* head() const { return lst_; }
103
104     root_list();
105     root_list(root_list const&);
106     root_list& operator = (const root_list&);
107
108   protected:
109
110     static E* next_get(E* p) {
111       assert(p != 0);
112       assert((*p).is_root());
113       return (*p).next_root_get();
114     }
115     static E* prev_get(E* p) {
116       assert(p != 0);
117       assert((*p).is_root());
118       return (*p).prev_root_get();
119     }
120     static void next_set(E* p, E* r) {
121       assert(p != 0); assert((*p).is_root());
122       (*p).next_root_set(r);
123       if(bidir && r != 0) (*r).prev_root_set(p);
124     }
125     static void next_clear(E* p) {
126       assert(p != 0); assert((*p).is_root());
127       (*p).next_root_set(0);
128     }
129     static void next_prev_clear(E* p) {
130       next_clear(p);
```

```
131         if(bidir) (*p).prev_root_set(0);
132       }
133       void head_set(E* p) {
134         (*this).lst_ = p;
135         if(bidir && p != 0 && !(*p).is_leaf())
136           (*p).prev_root_set(0); //???
137       }
138
139     public:
140
141       template<typename E2>
142       struct rebind_encapsulator_type {
143         typedef root_list<E2,bidir> other;
144       };
145
146       static const bool any_splice = false;
147
148       class adder_access_type {
149         root_list<E,bidir>& rl;
150
151       public:
152         adder_access_type(root_list<E,bidir>& rl) : rl(rl) {}
153
154         bool empty() { return rl.empty(); }
155
156         E* skip() { return rl.skip(); }
157         E* eject() { return rl.eject(); }
158       };
159       typedef adder_access_type & adder_access_ref;
160
161
162       typedef std::size_t size_type;
163
164       // structors
165
166       explicit root_list(E* head) {
167         head_set(head);
168         assert((*this).is_valid());
169         assert(!bidir);
170         // Complicated bidir code moved temporarely to root_list_bidir.hpp
171         }
172
173       // iterators
174
175       E* begin() const { return (*this).head(); }
176       E* end() const { return 0; }
177
178       // accessors
179
180       template <typename C>
181       E* find_top(C const& comparator) const {
182         E* y((*this).head());
183         if(y != 0)
184           for(E* x(next_get(y)); x != 0; x = next_get(x))
185             if(comparator((*y).element(), (*x).element())) y = x;
186         return y;
187       }
188
189       // properties
190
191       bool empty() const { return ((*this).head() == 0); }
192
193       int length() const {
194         int r(0);
195         for(E* p = (*this).head(); p != 0; p = next_get(p)) ++r;
196         return r;
197       };
198
199       bool is_valid() {
200         // Desc: validate linkages without using size info.
201         bool valid(true);
202         E* p((*this).head());
203         while(p != 0) {
204           valid &= (*p).is_valid();
```

```
205           E* q = next_get(p);
206           if(bidir && q != 0) {
207             valid &= prev_get(q) == p;
208           }
209           assert(valid);
210           p = q;
211         }
212         return valid;
213       }
214
215       template<typename BS // bit_scanner
216       > bool is_valid_bs(BS & bs) {
217         assert(is_valid());
218         bool heap_is_valid(true);
219         E * r((*this).head());
220         while(r != 0) {
221           typename E::height_type h2((*r).height());
222           heap_is_valid &= typename E::size_type(1 << h2) == bs.current();
223           bool is_root = (*r).is_root();
224           heap_is_valid &= is_root;
225           assert(heap_is_valid);
226           r = next_get(r);
227           bs.next_bit();
228         }
229         return heap_is_valid;
230       }
231
232       bool is_valid(size_type n) {
233         assert(is_valid());
234         E * r((*this).head());
235         typename E::height_type h(0);
236         bool heap_is_valid(true);
237         while(n != 0) {
238           if((n & 1) != 0) {
239             typename E::height_type h2((*r).height());
240             heap_is_valid &= h2 == h;
241             bool is_root = (*r).is_root();
242             heap_is_valid &= is_root;
243             assert(heap_is_valid);
244             r = next_get(r);
245           }
246           n >>= 1; ++h;
247         }
248         return heap_is_valid;
249       }
250
251       bool is_valid(size_type n) const {
252         return const_cast<root_list const*>(this).is_valid(n);
253       }
254
255       // modifiers
256
257       struct back_inserter {
258         E* last;
259         root_list& rl;
260         back_inserter(root_list& rl) : last(0), rl(rl) { assert(rl.empty()); }
261         void add(E* e) {
262           if(last == 0) rl.head_set(e);
263           else next_set(last, e);
264           last = e;
265         }
266       private:
267         back_inserter(back_inserter const&);
268         back_inserter& operator = (back_inserter const&);
269       };
270
271       void clear() { head_set(0); }
272
273       E* release() {
274         E* r((*this).head());
275         clear();
276         return r;
277       }
278
```

```
279       static E* inject(E* list, E* p) {
280         // Desc: add one root.
281         //assert((*this).head() == 0 || ((*p).height() < (*head()).height())); // binary
282         assert(list == 0 || ((*p).height() <= (*list).height())); // redundant
283         assert(next_get(p) == 0);
284         assert((*p).is_root());
285         next_set(p, list);
286         return p;
287       }
288
289       void inject(E* p) {
290         // Desc: add one root.
291         assert((*this).head() == 0 || ((*p).height() <= (*head()).height())); // redundant
292         assert(next_get(p) == 0);
293         assert((*p).is_root());
294         next_set(p, (*this).head());
295         (*this).head_set(p);
296       }
297
298       static E* inject_range(E* list, E* r, E* t) {
299         // Desc: add a root list in front.
300         assert((*r).is_root());
301         assert(next_get(t) == 0);
302         assert((list == 0) || ((*t).height() < (*list).height()));
303         next_set(t, list);
304         return r;
305       }
306
307       void inject_range(E* r, E* t) {
308         // Desc: add a root list.
309         assert((*r).is_root());
310         assert(next_get(t) == 0);
311         assert((*this).head() == 0 || ((*t).height() < (*head()).height()));
312         next_set(t, (*this).head());
313         (*this).head_set(r);
314       }
315
316       E* eject() {
317         assert((*this).head() != 0);
318         assertion_help(int l = length());
319         E* p((*this).head());
320
321         (*this).head_set(next_get((*this).head()));
322
323         next_clear(p);
324         assert((*p).is_root());
325         assert(length() == (l-1));
326         return p;
327       }
328
329       E* skip() {
330         assert((*this).head() != 0);
331         E* p((*this).head()); // (no forward clear)
332         (*this).lst_ = next_get((*this).head()); // (no backward clear)
333         return p;
334       }
335
336       void remove(E* x) {
337         assert((*x).is_root());
338         assert((*this).head() != 0);
339         if(x == (*this).head()) {
340           (*this).head_set(next_get((*this).head()));
341         } else {
342           E* w((*this).head());
343           assert(!bidir);
344           while(next_get(w) != x) {
345             w = next_get(w); assert(w != 0);
346           }
347           next_set(w, next_get(x));
348         }
349         next_prev_clear(x);
350       }
351
352       void remove_after(E* z) {
```

```
353         assert((*this).head() != 0);
354         if(z == 0) {
355           E* x((*this).head());
356           assert((*x).is_root());
357           (*this).head_set(next_get(x));
358           next_clear(x);
359         } else {
360           E* x(next_get(z));
361           assert((*x).is_root());
362           next_set(z, next_get(x));
363           next_prev_clear(x);
364         }
365       }
366
367       void promoted_to_root(E* p, E* d) {
368         if(d == (*this).head()) {
369           (*this).head_set(p);
370         } else if(bidir)
371           assert(E::promote_is_root_list_neutral);
372         else { // replace_root(p,d);
373           if(E::promote_is_root_list_neutral) {
374             return; // indirect value
375           }
376           E* w((*this).head());
377           while(next_get(w) != d) { w = next_get(w); assert(w != 0); }
378           next_set(w, p);
379         }
380       }
381
382       void replace(E* x, E* y) {
383         if(bidir) {
384           assert(x == (*this).head()); // replace head is the only legal bidir replace.
385           assert(x != 0); assert((*x).is_root());
386           assert(y != 0); assert(!careful || (*y).is_root());
387           assert(x != y);
388
389           (*this).head_set(y);
390           next_set(y, next_get(x));
391           next_set(x, 0);
392         } else {
393           // assert x is in rootlist and y isnt.
394           assert((*this).head() != 0);
395           assert(x != 0); assert((*x).is_root());
396           assert(y != 0); assert(!careful || (*y).is_root());
397
398           if(x == (*this).head()) {
399             (*this).head_set(y);
400             next_set(y, next_get(x));
401           } else {
402             E* w((*this).head());
403             while(next_get(w) != x) {
404               w = next_get(w); assert(w != 0);
405             }
406             next_set(w, y);
407             next_set(y, next_get(x));
408           }
409           next_prev_clear(x);
410         }
411       }
412
413       void swap(root_list& l2) {
414         // Precondition: The comparators and allocators must be compatible.
415         std::swap(lst_, l2.lst_);
416       }
417     };
418   }
419   #endif
```

A.30 root_list_bidir.hpp

```
1   #ifndef __CPHSTL_ROOT_LIST_BIDIR_H__
2   #define __CPHSTL_ROOT_LIST_BIDIR_H__
3
```

```
4  /*
5  Desc: Vuillemin+Brown+Bruun style bidirectional root list.
6  Note: The first node has never a backward link to list handle
7        because this node is accessed in constant time too.
8  Auth: Asger Bruun 2010.
9  Ref:  Jean Vuillemin. A data structure for manipulating priority queues.
10       Communications of the ACM 21 (1978), 309-315.
11 Note: This is the bidirectional version.
12       The bidirectional version has temporarely branched from
13       "root_list.hpp" because serious trouble combining the
14       two versions in one source.
15 */
16
17 #include "root_list.hpp" // root_list_node
18
19 namespace cphstl {
20   using namespace pqfw_node;
21
22   template<typename E>
23   class root_list_bidir {
24     static const bool bidir = true;
25   public:
26     // !: typedef typename E::root_list_base root_list_base;
27     typedef typename E::root_list_base::template
28       rebind_encapsulator_type<E>::other root_list_base;
29
30     static const bool any_splice = true;
31
32   private:
33
34     E* blst_;
35
36     E* before_head() { // root list behaving as a list node.
37       E* r((E*)&blst_);
38       assert((void*)r == (void*)&((*r).next_));
39       // Above: require next as first field (better if test was static).
40       return r;
41     }
42     E* head() const { return (E*) (*const_cast<root_list_bidir&>(*this).before_head()).
43         next_; }
44
45     root_list_bidir(); // disable
46     root_list_bidir(root_list_bidir const&); // disable
47     root_list_bidir& operator = (const root_list_bidir&);
48
49   protected:
50
51     static E* next_get(E* p) {
52       assert(p != 0);
53       assert((*p).is_root());
54       return (*p).next_root_get();
55     }
56     static void next_set(E* p, E* r) {
57       assert(p != 0);
58       assert(!root_list_base::is_spooky(p));
59       assert(!root_list_base::is_spooky(r));
60       (*p).next_root_set(r);
61       if(bidir && r != 0 && (*r).prev_rootable())
62         (*r).prev_root_set(p);
63     }
64     static void prev_clear(E* p, E* old_prev) {
65       assert(p != 0);
66       assert(!root_list_base::is_spooky(old_prev));
67       if(bidir && (*p).prev_rootable()) {
68         assert((*p).prev_root_get() == old_prev);
69         (*p).prev_root_set(0);
70       }
71     }
72     static bool prev_is_valid(E* p, E* prev) {
73       assert(p != 0);
74       assert(!root_list_base::is_spooky(prev));
75       assert(prev != 0);
76       if(!bidir) return true;
77       else {
```

```
77          bool valid;
78          if((*p).prev_rootable()) {
79            valid = (*p).prev_root_get() == prev;
80          } else valid = true;
81          assert(valid);
82          return valid;
83        }
84
85      }
86      void head_set(E* p) {
87        next_set((*this).before_head(),p);
88      }
89
90    public:
91
92      template<typename E2>
93      struct rebind_encapsulator_type {
94        typedef root_list_bidir<E2> other;
95      };
96
97      class adder_access_type {
98        // Desc: operation mode needed for adder on bidir list.
99        E* head_;
100
101     public:
102       adder_access_type(root_list_bidir& rl) : head_(rl.head()) {
103         if((*this).head_ != 0)
104           prev_clear((*this).head_, rl.before_head());
105         rl.clear();
106       }
107
108       bool empty() { return ((*this).head_ == 0); }
109
110       E* skip() {
111         assert((*this).head_ != 0);
112         E* p((*this).head_);
113         head_ = next_get((*this).head_);
114         return p;
115       }
116       E* eject() {
117         assert((*this).head_ != 0);
118         E* p((*this).head_);
119         (*this).head_ = next_get((*this).head_);
120         if((*this).head_ != 0) {
121           prev_clear((*this).head_,p);
122           next_set(p, 0);
123         }
124         assert((*p).is_root());
125         return p;
126       }
127     };
128     typedef adder_access_type & adder_access_ref;
129
130     typedef std::size_t size_type;
131
132     // structors
133
134     explicit root_list_bidir(E* head) : blst_(head) {
135       head_set(head);
136       assert((*this).is_valid());
137     }
138
139     // iterators
140
141     E* begin() const { return (*this).head(); }
142     E* end() const { return 0; }
143
144     // accessors
145
146     template <typename C>
147     E* find_top(C const& comparator) {
148       E* y((*this).head());
149       if(y != 0)
150         for(E* x(next_get(y)); x != 0; x = next_get(x))
```

```
151             if(comparator((*y).element(), (*x).element())) y = x;
152       return y;
153     }
154
155     // properties
156
157     bool empty() const { return ((*this).head() == 0); }
158
159     int length() const {
160       int r(0);
161       for(E* p = (*this).head(); p != 0; p = next_get(p)) ++r;
162       return r;
163     };
164
165     bool is_valid() {
166       // Desc: validate linkages without using size info.
167       bool valid(true);
168       E* p((*this).before_head());
169       E* q((E*) (*p).next());
170       while(p != 0 && q != 0) {
171         valid &= prev_is_valid(q,p);
172         assert(valid);
173         p = q;
174         q = next_get(q);
175       }
176       return valid;
177     }
178
179     template<typename BS // bit_scanner
180     > bool is_valid_bs(BS & bs) {
181       bool heap_is_valid((*this).is_valid());
182       assert(heap_is_valid);
183       E * r((*this).head());
184       while(r != 0) {
185         typename E::height_type h2((*r).height());
186         heap_is_valid &= typename E::size_type(1 << h2) == bs.current();
187         bool is_root = (*r).is_root();
188         heap_is_valid &= is_root;
189         assert(heap_is_valid);
190         r = next_get(r);
191         bs.next_bit();
192       }
193       return heap_is_valid;
194     }
195
196     bool is_valid(size_type n) {
197       E * r((*this).head());
198       typename E::height_type h(0);
199       bool heap_is_valid((*this).is_valid());
200       assert(heap_is_valid);
201       while(n != 0) {
202         if((n & 1) != 0) {
203           typename E::height_type h2((*r).height());
204           heap_is_valid &= h2 == h;
205           bool is_root = (*r).is_root();
206           heap_is_valid &= is_root;
207           assert(heap_is_valid);
208           r = next_get(r);
209         }
210         n >>= 1; ++h;
211       }
212       return heap_is_valid;
213     }
214
215     bool is_valid(size_type n) const {
216       return const_cast<root_list_bidir *>(this).is_valid(n);
217     }
218
219     // modifiers
220
221     struct back_inserter {
222       E* last;
223       root_list_bidir<E>& rl;
224       back_inserter(root_list_bidir<E>& rl) : last(0), rl(rl) { assert(rl.empty()); }
```

```
225         void add(E* e) {
226           if(last == 0) rl.head_set(e);
227           else { // non first digit has height > 0.
228             next_set(last, e);
229           }
230           last = e;
231           assert(rl.is_valid());
232         }
233       private:
234         back_inserter(back_inserter const&);
235         back_inserter& operator = (back_inserter const&);
236       };
237
238       void clear() {
239         (*this).head_set(0);
240       }
241
242       E* release() {
243         // Desc: remove head back link if bidir.
244         E* r((*this).head());
245         prev_clear((*this).head(),(*this).before_head()); //p, old_prev
246         clear();
247         return r;
248       }
249
250       static E* inject(E* list, E* p) {
251         // Desc: add one root.
252         assert(list == 0 || ((*p).height() <= (*list).height())); // redundant
253         assert(next_get(p) == 0);
254         assert((*p).is_root());
255         next_set(p, list);
256         return p;
257       }
258
259       void inject(E* p) {
260         // Desc: add one root.
261         assert((*this).head() == 0 || ((*p).height() <= (*(*this).head()).height())); //
               redundant
262         assert(next_get(p) == 0);
263         assert((*p).is_root());
264         next_set(p, (*this).head()); //  !!:
265         (*this).head_set(p);
266       }
267
268       static E* inject_range(E* list, E* r, E* t) {
269         // Desc: add a root list in front.
270         assert((*r).is_root());
271         assert(next_get(t) == 0);
272         assert((list == 0) || ((*t).height() < (*list).height()));
273         next_set(t, list);
274         return r;
275       }
276
277       void inject_range(E* r, E* t) {
278         // Desc: add a root list in front.
279         assert((*r).is_root());
280         assert(next_get(t) == 0);
281         assert((*this).empty() || ((*t).height() < (*(*this).head()).height()));
282         next_set(t, (*this).head());
283         next_set(before_head(),r);
284       }
285
286       E* eject() {
287         assert((*this).head() != 0);
288         assertion_help(int l = length());
289         E* p((*this).head());
290
291         (*this).head_set(next_get((*this).head()));
292
293         next_set(p, 0);
294         prev_clear(p,before_head());
295         assert((*p).is_root());
296         assert(length() == (l-1));
297         return p;
```

```
298       }
299
300       // No: E* skip();
301
302       void remove(E* x) {
303         assert((*x).is_root());
304         assert((*this).head() != 0);
305
306         // optimised fat join schedule bidirectional root list.
307         assert((*x).prev_rootable() || (x == (*this).head())
308           || (x == next_get((*this).head())) );
309
310         if((*x).prev_rootable()) {
311           E* p((*x).prev_root_get()); assert(p != 0);
312           E* n((*x).next_root_get());
313           (*x).prev_root_set(0);
314           (*p).next_root_set(n);
315           if(n != 0) {
316             (*x).next_root_set(0);
317             (*n).prev_root_set(p);
318           }
319         } else if(x == (*this).head()) {
320           E* p((*this).before_head());
321           E* n((*x).next_root_get());
322           (*p).next_root_set(n);
323           if(n != 0) {
324             next_set(x, 0);
325             if((*n).prev_rootable())
326               (*n).prev_root_set(p);
327           }
328         } else {
329           assert(x == next_get((*this).head()));
330           E* p((*this).head());
331           E* n((*x).next_root_get());
332           (*p).next_root_set(n);
333           if(n != 0) {
334             next_set(x, 0);
335             (*n).prev_root_set(p);
336           }
337         }
338       }
339
340       void remove_after(E* z) {
341         assert((*this).head() != 0);
342         if(z == 0) {
343           E* x((*this).head());
344           assert((*x).is_root());
345           (*this).head_set(next_get(x));
346           next_set(x, 0);
347           prev_clear(x,before_head());
348         } else {
349           E* x(next_get(z));
350           assert((*x).is_root());
351           next_set(z, next_get(x));
352           next_set(x, 0);
353           prev_clear(x,z);
354         }
355       }
356
357       void promoted_to_root(E*, E*) {
358         assert(E::promote_is_root_list_neutral);
359       }
360
361       // No: void replace_root(E* p, E* d);
362
363       void replace(E* x, E* y) {
364         assert(x == (*this).head()); // replace head is the only legal replace on bidir list.
365         assert(x != 0); assert((*x).is_root());
366         assert(y != 0); assert(!careful || (*y).is_root());
367         assert(x != y);
368
369         (*this).head_set(y);
370         next_set(y, next_get(x));
371         next_set(x, 0);
```

```
372       }
373
374       void swap(root_list_bidir<E>& l2) {
375         // Precondition: The comparators and allocators must be compatible.
376         assert((*this).is_valid());
377         assert(l2.is_valid());
378
379         E* tmp((*this).head());
380         if(tmp != 0) prev_clear((*this).head(),(*this).before_head());
381         if(l2.head() != 0) prev_clear(l2.head(),l2.before_head());
382         next_set((*this).before_head(),l2.head());
383         next_set(l2.before_head(),tmp);
384
385         assert((*this).is_valid());
386         assert(l2.is_valid());
387       }
388     };
389   }
390   #endif
```

B Code/CPHSTL_modified

B.1 assert.h++

```
/*
This is a version modified by Asger for binomial queue and msvc!!!
Date of snapshot = 20091012.

The assert macro is taken quite directly from the book [Steve
Maguire, Writing solid code, Microsoft Press (1993)]. This macro is
a statement that cannot be used in expression context.

The static assert is now provided by the compiler.

Author: Jyrki Katajainen 2001, 2008
*/

#ifndef __CPHSTL_ASSERT__
#define __CPHSTL_ASSERT__

#include <cstdio> // std::fflush, std::fprintf
#include <cstdlib> // std::abort

#ifdef NDEBUG
#define assert(condition)
#define assertion_help(h)
#define __break_here
#else

// put this around assertion code to avoid warnings when NDEBUG.
#define assertion_help(h) h

static long long current_break_point_number = 0;
// usage: assert(++current_break_point_number != 177);
//     or: assertion_help(if(++current_break_point_number == 177) __break_here; )



#ifdef _MSC_VER

#include <iostream>

  bool __assert(bool e, char const* message, char const* strFile, char const* strFunc,
      unsigned int uLine) {
    if(!e) {
      std::fflush(NULL);
      std::fprintf(stderr, "\nAssertion '%s' failed: %s, line %u, function %s\n",
        message, strFile, uLine, strFunc);
      std::fflush(stderr);
      //std::abort(); // optional
    }
    return e;
  }

  template <typename V1, typename V2>
  bool __assert_equal(V1 e, V2 a, char const* s1, char const* s2, char const* strFile,
      char const* strFunc, unsigned int uLine) {
    // displays the actual data values (useful when debugging).
    if(e != a) {
      std::cout.flush();
      std::cerr << "\nAssertion '" << s1 << " == " << s2 << "' failed (" << e << " != " << a
          << "): "
            << strFile << ", line " << uLine << ", function " << strFunc << "\n";
      std::cerr.flush();
      //std::abort(); // optional
    }
    return e == a;
  }


#ifdef _M_X64
  #define __break_here std::abort()
#else
```

```
67    #define __break_here __asm { int 3 } // invoke debugger break point
68  #endif
69
70  #define assert(e) { \
71    __analysis_assume(e); \
72    if (!(__assert(e, #e, __FILE__, __FUNCTION__, __LINE__))) \
73      __break_here; \
74  }
75  #define assert_equal(e,a) { \
76    __analysis_assume(e); __analysis_assume(a); \
77    if (!(__assert_equal(e, a, #e, #a, __FILE__, __FUNCTION__, __LINE__))) \
78      __break_here; \
79  }
80
81  #elif defined(__GNUC__)
82
83  #include <signal.h>
84
85  #define __break_here raise(SIGINT)
86
87    void __assert(char const* message, char const* strFile, unsigned int uLine) {
88      std::fflush(NULL);
89      std::fprintf(stderr, "\nAssertion‘%sÂ´failed:%s,line%u\n",
90        message, strFile, uLine);
91      std::fflush(stderr);
92      //__break_here;
93      std::abort();
94    }
95
96  #define assert(condition) \
97    if (condition) { \
98    } \
99    else \
100   __assert(#condition, __FILE__, __LINE__)
101
102
103 #else
104
105   void __assert(char const* message, char const* strFile, unsigned int uLine) {
106     std::fflush(NULL);
107     std::fprintf(stderr, "\nAssertion‘%sÂ´failed:%s,line%u\n",
108       message, strFile, uLine);
109     std::fflush(stderr);
110     std::abort();
111   }
112
113 #define assert(condition) \
114   if (condition) { \
115   } \
116   else \
117   cphstl::__assert(#condition, __FILE__, __LINE__)
118
119 #endif
120
121
122 #endif
123 #endif
```

B.2 stl-meldable-priority-queue.h++

```
1   /*
2     A meldable priority queue is a container which provides forward
3     iterators to the elements stored.
4
5     CPH STL guarantees:
6
7     1) Member function push() returns an iterator (or a handle) to the
8     given element and this iterator remains valid the whole life time of
9     the element.
10
11    2) Iterator operations take logarithmic time in the worst case.
12
13    3) Member function top() is a constant-time operation.
```

```
14
15    4) Member functions push(), pop(), erase(), and increase() have the
16    logarithmic worst-case cost.
17
18    5) Member function meld() is relatively fast having a
19    polylogarithmic worst-case cost.
20
21    6) The data structure uses a linear number of words in addition to
22    the elements stored.
23
24    Container requirements not fulfilled [C++ standard Â§23]:
25
26    7) For an iterator p, expressions --p and p-- are not supported.
27
28    8) For two meldable priority queues a and b, the following expressions
29    are not supported: a == b, a != b, a < b, a > b, a <= b, and a >= b.
30
31    Author: Jyrki Katajainen, 2005, 2006, 2009, 2010
32  */
33
34  // 2010 Asger Bruun:
35  // 1) Added automatic coupling to optional realisator members.
36  //    The only change nescesarry at declaration level was adding
37  //    a protected access method passable to implementation policies
38  //    (alternativey to a bunch of friend declarations).
39  // 2) Replaced the template arguments by a compact design.
40  // 3) Added converting templates for backward compatibility.
41
42  #ifndef _CPHSTL_MELDABLE_PRIORITY_QUEUE_
43  #define _CPHSTL_MELDABLE_PRIORITY_QUEUE_
44
45  #include <cstddef> // std::size_t and std::ptrdiff_t
46  #include <functional> // std::less
47  #include <memory> // std::allocator
48  #include "multiple-heap-framework.h++"
49  #include "priority-queue-iterator.h++"
50  #include "weak-heap-node.h++"
51
52
53  namespace cphstl {
54    template <typename CFG>
55    class meldable_priority_queue_alt {
56    public:
57
58      // types
59
60      typedef typename CFG::V value_type;
61      typedef typename CFG::C comparator_type;
62      typedef typename CFG::A allocator_type;
63      typedef typename CFG::E encapsulator_type;
64      typedef typename CFG::R realizator_type;
65      typedef typename CFG::I iterator;
66      typedef typename CFG::J const_iterator;
67      typedef typename CFG::R::reference reference;
68      typedef typename CFG::R::const_reference const_reference;
69      typedef typename CFG::V* pointer;
70      typedef typename CFG::V const* const_pointer;
71      typedef std::size_t size_type;
72      typedef std::ptrdiff_t difference_type;
73      typedef std::reverse_iterator<iterator> reverse_iterator;
74      typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
75      typedef meldable_priority_queue_alt<CFG> container_type;
76
77    protected:
78
79      typedef typename CFG::A::template
80        rebind<typename CFG::E>::other encapsulator_allocator_type;
81
82      encapsulator_allocator_type allocator;
83      realizator_type realizator;
84
85      encapsulator_type* create(value_type const&);
86      void destroy(encapsulator_type*);
87
```

```
88      template <typename K>
89      void insert(K, K);
90
91      struct protected_access {
92        // Note: Passes protected access to non friend classes.
93        typedef container_type mpq_type;
94        container_type* mpq;
95
96        protected_access(container_type* mpq): mpq(mpq) {};
97
98        container_type const* get_mpq() const { return mpq; }
99        container_type* get_mpq() { return mpq; }
100       realizator_type const& get_realizator() const { return (*mpq).realizator; }
101       realizator_type& get_realizator() { return (*mpq).realizator; }
102
103       encapsulator_type* create(value_type const& v) { return (*mpq).create(v); }
104       void destroy(encapsulator_type* p) { (*mpq).destroy(p); }
105       template <typename K>
106       void insert(K begin, K end) { (*mpq).insert(begin, end); }
107     };
108
109   public:
110
111     // structors
112
113     explicit meldable_priority_queue_alt(
114       comparator_type const& = comparator_type()
115       , allocator_type const& = allocator_type()
116     );
117     meldable_priority_queue_alt(meldable_priority_queue_alt const&);
118     meldable_priority_queue_alt& operator=(meldable_priority_queue_alt const&);
119     ~meldable_priority_queue_alt();
120
121     // iterators
122
123     iterator begin();
124     const_iterator begin() const;
125     iterator end();
126     const_iterator end() const;
127
128     // accessors
129
130     allocator_type get_allocator() const;
131     comparator_type get_comparator() const;
132     bool empty() const;
133     size_type size() const;
134     size_type max_size() const;
135     const_iterator top() const;
136
137     // modifiers
138
139     iterator top();
140     iterator push(value_type const&);
141     void pop();
142     void erase(iterator);
143     void increase(iterator, value_type const&);
144     void clear(); // default or custom
145     void meld(meldable_priority_queue_alt&);
146     void swap(meldable_priority_queue_alt&);
147
148 #ifdef DEBUG
149
150     bool is_valid() {
151       return realizator.is_valid();
152     }
153
154     void show() {
155       realizator.show();
156     }
157
158 #endif
159
160   };
161
```

```
162     // algorithms
163
164     template <typename CFG>
165     meldable_priority_queue_alt<CFG>&
166     meld(meldable_priority_queue_alt<CFG>&,
167          meldable_priority_queue_alt<CFG>&);
168
169     template <typename CFG>
170     void swap(meldable_priority_queue_alt<CFG>&,
171         meldable_priority_queue_alt<CFG>&);
172   }
173
174
175   namespace cphstl { // --- Backward compatibility ---
176
177     // Convert old style direct heap store template to new compact format.
178     template <
179       typename V0, typename C0, typename A0, typename E0
180       , typename R0, typename I0, typename J0
181     >
182     struct meldable_priority_queue_config {
183       typedef V0 V; typedef C0 C; typedef A0 A; typedef E0 E;
184       typedef R0 R; typedef I0 I; typedef J0 J;
185     };
186
187     // Original style direct heap store template.
188     template <
189       typename V,
190       typename C = std::less<V>,
191       typename A = std::allocator<V>,
192       typename E = cphstl::weak_heap_node<V, A>,
193       typename R = cphstl::multiple_heap_framework<V, C, A, E>,
194       typename I = cphstl::priority_queue_iterator<E, R, false>,
195       typename J = cphstl::priority_queue_iterator<E, R, true>
196     >
197     class meldable_priority_queue
198       : public meldable_priority_queue_alt<
199         meldable_priority_queue_config<V,C,A,E,R,I,J>
200       >
201     {
202     public:
203       meldable_priority_queue()
204         : meldable_priority_queue_alt<
205           meldable_priority_queue_config<V,C,A,E,R,I,J>
206         >() {}
207     };
208   }
209
210   #include "stl-meldable-priority-queue.i++"
211   #endif
```

B.3 stl-meldable-priority-queue.i++

```
1   /*
2   A meldable priority queue is a container class that just calls the
3   functions available in the realizator class.
4
5   Author: Jyrki Katajainen Â© 2006, 2009, 2010
6   */
7
8   #include "stl-meldable-priority-queue.h++"
9
10  // 2010 Asger Bruun:
11  // Added automatic coupling to optional realisator members.
12  // using static custom method selection.
13  //
14  // Compile time realisator constants:
15  //    bool cphstl::has_member_empty<T>::value
16  //    bool cphstl::has_member_clear<T>::value
17  //    bool cphstl::has_member_meld<T>::value
18  //    bool cphstl::has_member_clone<T>::value
19  //    bool cphstl::has_member_pop<T>::value
20
```

```
21 #include "type.h++" // cphstl::bool2type
22 #include "has_member.hpp" // ab static member testing macros.
23
24 DEFINE_STATIC_TEST_HAS_MEMBER(empty)
25 DEFINE_STATIC_TEST_HAS_MEMBER(clear)
26 DEFINE_STATIC_TEST_HAS_MEMBER(meld)
27 DEFINE_STATIC_TEST_HAS_MEMBER(clone)
28 DEFINE_STATIC_TEST_HAS_MEMBER(pop)
29
30 namespace cphstl {
31   namespace { // method selectors.
32     template<typename R, bool custom_impl>
33     struct select_empty { // no custom
34       static bool empty(R const& r) {
35         return r.size()==0;
36       }
37     };
38     template<typename R>
39     struct select_empty<R,true> { // custom
40       static bool empty(R const& r) {
41         return r.empty();
42       }
43     };
44
45     template<typename PA, bool custom_impl>
46     struct select_clear { // no custom
47       static void clear(PA& pa) {
48         typedef typename PA::mpq_type Q;
49         typename Q::encapsulator_type* p;
50         typename Q::realizator_type& r(pa.get_realizator());
51         while (!r.empty()) {
52           p = r.extract();
53           pa.destroy(p);
54         }
55       }
56     };
57     template<typename PA>
58     struct select_clear<PA,true> { // custom
59       static void clear(PA& pa) {
60         pa.get_realizator().clear(pa);
61       }
62     };
63
64     template<typename PA, bool custom_impl>
65     struct select_meld { // no custom
66       static void meld(PA& q1, PA& q2) {
67         while(!(*q2.mpq).empty()) {
68           (*q1.mpq).push(*(*q2.mpq).top());
69           (*q2.mpq).pop();
70         }
71       }
72     };
73     template<typename PA>
74     struct select_meld<PA,true> { // custom
75       static void meld(PA& q1, PA& q2) {
76         q1.get_realizator().meld(q2.get_realizator());
77       }
78     };
79
80     template<typename PA, bool custom_impl>
81     struct select_clone { // no custom
82       static void clone(PA& q1, PA const& q2) {
83         q1.insert((*(q2.get_mpq())).begin(), (*(q2.get_mpq())).end());
84       }
85     };
86     template<typename PA>
87     struct select_clone<PA,true> { // custom
88       static void clone(PA& q1, PA const& q2) {
89         q1.get_realizator().clone(q2.get_realizator(),q1);
90       }
91     };
92
93     template<typename PA, bool custom_impl>
94     struct select_pop { // no custom
```

```
95        static void pop(PA& pa) {
96          typedef typename PA::mpq_type Q;
97          typename Q::realizator_type& r(pa.get_realizator());
98          typename Q::encapsulator_type* p(r.top());
99          r.extract(p);
100         pa.destroy(p);
101
102       }
103     };
104     template<typename PA>
105     struct select_pop<PA,true> { // custom
106       static void pop(PA& pa) {
107         pa.get_realizator().pop(pa);
108       }
109     };
110   }
111 }
112
113 namespace cphstl {
114   template <typename CFG>
115   typename meldable_priority_queue_alt<CFG>::encapsulator_type*
116     meldable_priority_queue_alt<CFG>::create(value_type const& v) {
117       encapsulator_type* p = allocator.allocate(1);
118       try {
119         new (p) encapsulator_type(v, allocator);
120       }
121       catch (...) {
122         destroy(p);
123         throw;
124       }
125       return p;
126   }
127
128   template <typename CFG>
129   void
130     meldable_priority_queue_alt<CFG>::destroy(encapsulator_type* p) {
131       p->~encapsulator_type();
132       allocator.deallocate(p, 1);
133   }
134
135   template <typename CFG>
136   template <typename K>
137   void
138     meldable_priority_queue_alt<CFG>::insert(K b, K e) {
139       meldable_priority_queue_alt q;
140       encapsulator_type* d(0);
141       try {
142         for (K c = b; c != e; ++c) {
143           d = create(*c);
144           q.realizator.insert(d);
145         }
146       }
147       catch (...) {
148         destroy(d);
149         q.clear();
150         throw;
151       }
152       meld(q);
153   }
154
155   template <typename CFG>
156   meldable_priority_queue_alt<CFG>::meldable_priority_queue_alt(
157     comparator_type const& comparator, allocator_type const& allocator)
158     : realizator(comparator, allocator) {
159   }
160
161   template <typename CFG>
162   meldable_priority_queue_alt<CFG>::meldable_priority_queue_alt(
163     meldable_priority_queue_alt const& other
164     ) : realizator(comparator_type(),allocator)
165   {
166     const bool custom_impl = has_member_clone<realizator_type>::value;
167     protected_access pa1(this);
168     protected_access pa2(& const_cast<meldable_priority_queue_alt &>(other));
```

```
169       select_clone<protected_access, custom_impl>::clone(pa1, pa2);
170       //meldable_priority_queue_alt q(comparator_type(),allocator);
171       //try {
172       //  q.insert(other.begin(), other.end());
173       //}
174       //catch (...) {
175       //  while (q.size() != 0) {
176       //    encapsulator_type* d = q.realizator.extract();
177       //    destroy(d);
178       //  }
179       //  throw;
180       //}
181       //(*this).swap(q);
182     }
183
184     template <typename CFG>
185     typename meldable_priority_queue_alt<CFG>::container_type&
186       meldable_priority_queue_alt<CFG>::operator=(
187       meldable_priority_queue_alt const& other) {
188         meldable_priority_queue_alt q(other);
189         (*this).swap(q);
190         q.clear();
191         return *this;
192
193         //meldable_priority_queue_alt q;
194         //try {
195         //  q.insert(other.begin(), other.end());
196         //}
197         //catch (...) {
198         //  while (q.size() != 0) {
199         //    encapsulator_type* d = q.realizator.extract();
200         //    destroy(d);
201         //  }
202         //  throw;
203         //}
204         //(*this).swap(q);
205         //while (q.size() != 0) {
206         //  encapsulator_type* d = q.realizator.extract();
207         //  destroy(d);
208         //}
209         //return *this;
210     }
211
212     template <typename CFG>
213     meldable_priority_queue_alt<CFG>::~meldable_priority_queue_alt() {
214       clear();
215     }
216
217     template <typename CFG>
218     typename meldable_priority_queue_alt<CFG>::allocator_type
219       meldable_priority_queue_alt<CFG>::get_allocator() const {
220         return realizator.get_allocator();
221     }
222
223     template <typename CFG>
224     typename meldable_priority_queue_alt<CFG>::comparator_type
225       meldable_priority_queue_alt<CFG>::get_comparator() const {
226         return realizator.get_comparator();
227     }
228
229     template <typename CFG>
230     typename meldable_priority_queue_alt<CFG>::iterator
231       meldable_priority_queue_alt<CFG>::begin() {
232         return iterator(realizator.begin(), &realizator);
233     }
234
235     template <typename CFG>
236     typename meldable_priority_queue_alt<CFG>::const_iterator
237       meldable_priority_queue_alt<CFG>::begin() const {
238         return const_iterator(realizator.begin(), (realizator_type*) &realizator);
239     }
240
241     template <typename CFG>
242     typename meldable_priority_queue_alt<CFG>::iterator
```

```
243       meldable_priority_queue_alt<CFG>::end() {
244         return iterator(realizator.end(), &realizator);
245     }
246
247     template <typename CFG>
248     typename meldable_priority_queue_alt<CFG>::const_iterator
249       meldable_priority_queue_alt<CFG>::end() const {
250         return const_iterator(realizator.end(), (realizator_type*) &realizator);
251     }
252
253     template <typename CFG>
254     bool
255       meldable_priority_queue_alt<CFG>::empty() const {
256         const bool custom_impl = has_member_empty<realizator_type>::value;
257         return select_empty<realizator_type, custom_impl>::empty((*this).realizator);
258         //return (*this).size() == size_type(0); // !: design flaw
259     }
260
261     template <typename CFG>
262     typename meldable_priority_queue_alt<CFG>::size_type
263       meldable_priority_queue_alt<CFG>::size() const {
264         return realizator.size();
265     }
266
267     template <typename CFG>
268     typename meldable_priority_queue_alt<CFG>::size_type
269       meldable_priority_queue_alt<CFG>::max_size() const {
270         return realizator.max_size();
271     }
272
273     template <typename CFG>
274     typename meldable_priority_queue_alt<CFG>::const_iterator
275       meldable_priority_queue_alt<CFG>::top() const {
276         return const_iterator(realizator.top(), &realizator);
277     }
278
279     template <typename CFG>
280     typename meldable_priority_queue_alt<CFG>::iterator
281       meldable_priority_queue_alt<CFG>::top() {
282         return iterator(realizator.top(), &realizator);
283     }
284
285     template <typename CFG>
286     typename meldable_priority_queue_alt<CFG>::iterator
287       meldable_priority_queue_alt<CFG>::push(value_type const& v) {
288         encapsulator_type* p = create(v);
289         realizator.insert(p);
290         return iterator(p, &realizator);
291     }
292
293     template <typename CFG>
294     void
295       meldable_priority_queue_alt<CFG>::pop() {
296         const bool custom_impl = has_member_pop<realizator_type>::value;
297         protected_access pa(this);
298         select_pop<protected_access, custom_impl>::pop(pa);
299
300         //encapsulator_type* p = realizator.top();
301         //realizator.extract(p);
302         //destroy(p);
303     }
304
305     template <typename CFG>
306     void
307       meldable_priority_queue_alt<CFG>::erase(iterator t) {
308         encapsulator_type* p = (encapsulator_type*) t;
309         realizator.extract(p);
310         destroy(p);
311     }
312
313     template <typename CFG>
314     void
315       meldable_priority_queue_alt<CFG>::increase(iterator t, value_type const& v) {
316         encapsulator_type* p = (encapsulator_type*) t;
```

```
317         realizator.increase(p, v);
318       }
319
320       template <typename CFG>
321       void
322         meldable_priority_queue_alt<CFG>::clear() {
323           const bool custom_impl = has_member_clear<realizator_type>::value;
324           protected_access pa(this);
325           select_clear<protected_access, custom_impl>::clear(pa);
326
327           //select_clear<container_type, realizator_type, protected_access, custom_impl>
328           //  ::clear((*this), (*this).realizator, pa);
329
330           //while (realizator.size() != 0) {
331           //  encapsulator_type* p = realizator.extract();
332           //  destroy(p);
333           //}
334       }
335
336       template <typename CFG>
337       void
338         meldable_priority_queue_alt<CFG>::meld(meldable_priority_queue_alt& q) {
339           const bool custom_impl = has_member_meld<realizator_type>::value;
340           protected_access pa1(this), pa2(&q);
341           select_meld<protected_access, custom_impl>::meld(pa1, pa2);
342           //    realizator.meld(q.realizator);
343       }
344
345       template <typename CFG>
346       void
347         meldable_priority_queue_alt<CFG>::swap(meldable_priority_queue_alt& q) {
348           realizator.swap(q.realizator);
349       }
350
351       template <typename CFG>
352       meldable_priority_queue_alt<CFG>& meld(
353         meldable_priority_queue_alt<CFG>& r,
354         meldable_priority_queue_alt<CFG>& s
355         )
356       {
357           r.meld(s.realizator);
358           return r;
359       }
360
361       template <typename CFG>
362       void swap(meldable_priority_queue_alt<CFG>& r,
363         meldable_priority_queue_alt<CFG>& s) {
364           r.swap(s.realizator);
365       }
366   }
```

C Benchmarking

C.1 benchmarking.hpp

```
#ifndef _CPHSTL_BENCHMARKING_HPP_
#define _CPHSTL_BENCHMARKING_HPP_
/*
* Desc: Repeated measurements of resource consumption.
*       An attempt to simplify measurements and to improve
*       accuracy in time, includes some basic statistics.
*
* Auth: Asger Bruun 2009-2010
*
* Impl: time_counter - counts system time ticks.
*       compare_counter - counts compares.
*       alloc_counter - counts allocations.
*       stop_watch - time_counter converted to seconds.
*       combi_counter - the above counters combined.
*
*       counting_compare - wrapper, makes your comparator count.
*       counting_allocator - counting allocator replacement.
*
*       cpu_attention - get optimal system priority for timing.
*/

#include "assert.h++"
#include "ms_c_fix.hpp"

#include "benchmark_statistics.hpp"
#include "benchmark_compare_counter.hpp"
#include "benchmark_alloc_counter.hpp"
#include "benchmark_time_counter.hpp"
#include "benchmark_faster_clock_policy.hpp"
#include "benchmark_tsc_clock_policy.hpp"
#include "benchmark_cpu_attention.hpp"
#include "benchmark_combi_counter.hpp"
#include "benchmark_profiling_counter.hpp"

#endif
```

C.2 benchmark_alloc_counter.hpp

```
#ifndef _CPHSTL_BENCHMARK_ALLOC_COUNTER_HPP_
#define _CPHSTL_BENCHMARK_ALLOC_COUNTER_HPP_
/*
Desc: An allocation counter, offering a counting allocator.
Auth: Asger Bruun 2009-2010.
Impl: alloc_counter - the performance counter.
:     counting_allocator - use this instead of the std::allocator to get the count.
*/
#include "benchmark_statistics.hpp"

namespace cphstl {
  namespace benchmarking {

    struct alloc_counter_base {
      typedef long long counter_type;
      static counter_type allocs;
      static counter_type current() { return allocs; }
    };
    alloc_counter_base::counter_type alloc_counter_base::allocs(0);

    struct dealloc_counter_base {
      typedef long long counter_type;
      static counter_type deallocs;
      static counter_type current() { return deallocs; }
    };
    dealloc_counter_base::counter_type dealloc_counter_base::deallocs(0);


    struct alloc_counter : public sampler<alloc_counter_base> {
```

```
30        sampler<dealloc_counter_base> counter2;
31
32        static const unsigned int multiplicity = 1;
33        static std::string unit_name(unsigned int counter_idx) {
34          switch(counter_idx) {
35            case 0: return  "allocated octets";
36            default: assert(0); return 0; break;
37          }
38        }
39
40
41        alloc_counter(bool single_op)
42          : sampler<alloc_counter_base>(single_op), counter2(single_op) {}
43
44
45        void start(size_type const& element_count) {
46          sampler<alloc_counter_base>::start(element_count);
47          counter2.start(element_count);
48        }
49        void stop() {
50          sampler<alloc_counter_base>::stop();
51          counter2.stop();
52        }
53
54        void report();
55
56        template<typename PP /* = data_columns::default_post_processor*/>
57        void gnuplot_data(std::string description);
58      };
59      void alloc_counter::report() {
60        // Note: the allocation counting is very simple,
61        //       and chould be extended with more figures.
62        for(element_counts_type::const_iterator i
63          = element_counts.begin(); i != element_counts.end(); ++i) {
64            counter2.current_ec = current_ec = (*i).first;
65
66            counter_fraction avg(average());
67            double rsd(relative_std_dev());
68
69            counter_fraction avg2(counter2.average());
70            double rsd2(counter2.relative_std_dev());
71
72            std::cout.precision(3);
73            std::cout << (i == element_counts.begin()?"allo ":"           ");
74            std::cout << current_ec << " + " << avg;
75            if(rsd != 0) std::cout << " (rsd " << rsd << "%)";
76            std::cout << " - " << avg2;
77            if(rsd2 != 0) std::cout << " (rsd " << rsd2 << "%)";
78            std::cout << " octets.\n";
79        }
80      }
81      template<typename PP /* = data_columns::default_post_processor*/>
82      void alloc_counter::gnuplot_data(std::string description) {
83        sampler<alloc_counter_base>::template gnuplot_data<PP>(description+" - "+unit_name(0));
84      }
85
86
87      template <typename T = int, typename A3 = std::allocator<T> >
88      class counting_allocator : public A3::template rebind<T>::other {
89        // Warn: quick and dirty wrapping of std::allocator.
90      public:
91        typedef typename A3::pointer pointer;
92        typedef typename A3::size_type size_type;
93        template<typename U>
94        struct rebind {
95          typedef counting_allocator<U, typename A3::template rebind<U>::other> other;
96        };
97
98        counting_allocator() : A3() {}
99        counting_allocator(A3 const& a) : A3(a) {}
100       template<typename U>
101       counting_allocator(counting_allocator<U> const&) {} //: A3(a) {}
102
103       pointer allocate(size_type cnt, const void* p = 0) {
```

```
104           alloc_counter_base::allocs+= sizeof(T) * cnt;
105           return A3::allocate(cnt, p);
106         }
107         void deallocate(pointer p, size_type s) {
108           dealloc_counter_base::deallocs+= sizeof(T) * s;
109           A3::deallocate(p, s);
110         }
111         bool operator == (counting_allocator const&) { return true; }
112         bool operator != (counting_allocator const& a) { return !operator == (a); }
113       };
114     }
115   }
116   #endif
```

C.3 benchmark_combi_counter.hpp

```
1   #ifndef _CPHSTL_BENCHMARK_COMBI_COUNTER_HPP_
2   #define _CPHSTL_BENCHMARK_COMBI_COUNTER_HPP_
3   /*
4   Desc: Combined time conversion, allocation and compare counting.
5   Auth: Asger Bruun 2009-2010.
6   */
7   #include <ctime>
8   #include "benchmark_statistics.hpp"
9
10  namespace cphstl {
11    namespace benchmarking {
12      struct stop_watch : public time_counter<> {
13        // Desc: convert time_counter to seconds.
14        typedef long double seconds;
15
16        stop_watch(bool single_op)
17          : time_counter<>(single_op) {}
18
19        static const unsigned int multiplicity = 2;
20        static std::string unit_name(unsigned int counter_idx) {
21          switch(counter_idx) {
22            case 0: return time_counter<>::unit_name(0);
23            case 1: return "seconds";
24            default: assert(0); return 0; break;
25          }
26        }
27
28        void report();
29        template<typename PP /* = data_columns::default_post_processor*/>
30        void gnuplot_data(std::string description);
31        seconds average_time();
32        seconds sample_std_dev_time();
33        std::vector<seconds> box_and_whispers_times();
34        std::vector<seconds> quartile_times();
35      };
36      void stop_watch::report() {
37        for(element_counts_type::const_iterator i
38          = element_counts.begin(); i != element_counts.end(); ++i) {
39            current_ec = (*i).first;
40            seconds avg(average_time());
41            double rsd(relative_std_dev());
42            double factor = avg / std::log2(double(current_ec));
43            std::cout.precision(3);
44            std::cout << (i == element_counts.begin()?"time ":"           ");
45            std::cout << current_ec << " " << avg
46              << ", rsd " << rsd << "%,\t lg.fac " << factor << " lg n.\n";
47        }
48      }
49      template<typename PP /* = data_columns::default_post_processor*/>
50      void stop_watch::gnuplot_data(std::string description) {
51        time_counter<>::gnuplot_data<PP>(description);
52        std::cout << "\n# " << description << " - " << unit_name(1) << "\n";
53        std::cout << "# count  pct5  Q1  median  Q3  pct95  min  max  avg  focus\n";
54        for(element_counts_type::const_iterator i
55          = element_counts.begin(); i != element_counts.end(); ++i) {
56            current_ec = (*i).first;
57            std::cout << current_ec;
```

```
58            std::vector<seconds> qt(box_and_whispers_times());
59            for(std::vector<seconds>::const_iterator j = qt.begin(); j != qt.end(); ++j) {
60              std::cout << "␣␣" << *j;
61            }
62            std::cout << "␣␣" << average_time() << "␣␣0.5\n";
63          }
64          std::cout << "\n";
65        }
66        stop_watch::seconds stop_watch::average_time() {
67          assert(average() > 0); assert(current_ec > 0);
68          return seconds(average()) / seconds(frequency());
69        }
70        stop_watch::seconds stop_watch::sample_std_dev_time() {
71          return seconds(sample_std_dev()) / seconds(frequency());
72        }
73        std::vector<stop_watch::seconds> stop_watch::box_and_whispers_times() {
74          std::vector<double> q(box_and_whispers());
75          std::vector<seconds> ret(q.size());
76          for(int i = 0; i != (5+2); ++i)
77            ret[i] = seconds(q[i])/seconds(frequency());
78          return ret;
79        }
80        std::vector<stop_watch::seconds> stop_watch::quartile_times() {
81          std::vector<double> q(quartiles());
82          std::vector<seconds> ret(q.size());
83          for(int i = 0; i != 5; ++i)
84            ret[i] = seconds(q[i])/seconds(frequency());
85          return ret;
86        }
87
88        struct combi_counter {
89          typedef std::size_t size_type;
90          typedef size_type counter_type;
91
92          stop_watch sw;
93          compare_counter cc;
94          alloc_counter ac;
95
96          combi_counter(bool single_op)
97            : sw(single_op), cc(single_op), ac(single_op) {}
98
99          void prepare(size_type const& elements, size_type const& repeats) {
100           ac.prepare(elements, repeats);
101           cc.prepare(elements, repeats);
102           sw.prepare(elements, repeats);
103         }
104         void start(size_type const& element_count) {
105           ac.start(element_count);
106           cc.start(element_count);
107           sw.start(element_count);
108         }
109         void stop() { sw.stop(); cc.stop(); ac.stop(); }
110         void finish() { ac.finish(); cc.finish(); sw.finish(); }
111
112         static const unsigned int multiplicity = stop_watch::multiplicity
113           + compare_counter::multiplicity + alloc_counter::multiplicity;
114
115         static std::string unit_name(unsigned int counter_idx) {
116           const unsigned int s(stop_watch::multiplicity)
117             , c(compare_counter::multiplicity), a(alloc_counter::multiplicity);
118           if(counter_idx<s)
119             return stop_watch::unit_name(counter_idx);
120           else if(counter_idx<(s + c))
121             return compare_counter::unit_name(counter_idx - s);
122           else if(counter_idx<(s + c + a))
123             return alloc_counter::unit_name(counter_idx - (s+c));
124           else { assert(0); return 0; }
125         }
126
127         void report() {
128           sw.report();
129           std::cout << "␣␣␣␣␣␣"; cc.report();
130           std::cout << "␣␣␣␣␣␣"; ac.report();
131         }
```

```
132
133       template<typename PP /* = data_columns::default_post_processor*/>
134       void gnuplot_data(std::string description) {
135         sw.gnuplot_data<PP>(description);
136         cc.gnuplot_data<PP>(description);
137         ac.gnuplot_data<PP>(description);
138       }
139     };
140   }
141 }
142 #endif
```

C.4 benchmark_compare_counter.hpp

```
1  #ifndef _CPHSTL_BENCHMARK_COMPARE_COUNTER_HPP_
2  #define _CPHSTL_BENCHMARK_COMPARE_COUNTER_HPP_
3  /*
4  Desc: A compare counter, combined with a counting compare.
5  Auth: Asger Bruun 2009-2010.
6  Impl: compare_counter - the performance counter.
7  :     counting_compare - wrap your compare with this to get the count.
8  */
9  #include "benchmark_statistics.hpp"
10
11 namespace cphstl {
12   namespace benchmarking {
13     struct compare_counter_base {
14       typedef long long counter_type;
15       static counter_type comps;
16       static counter_type current() { return comps; }
17     };
18     compare_counter_base::counter_type compare_counter_base::comps(0);
19
20     struct compare_counter : public sampler<compare_counter_base> {
21       static const unsigned int multiplicity = 1;
22       static std::string unit_name(unsigned int counter_idx) {
23         switch(counter_idx) {
24           case 0: return  "compares";
25           default: assert(0); return 0; break;
26         }
27       }
28
29       compare_counter(bool single_op) : sampler<compare_counter_base>(single_op) {}
30       void report();
31       template<typename PP /* = data_columns::default_post_processor*/>
32       void gnuplot_data(std::string description);
33     };
34     void compare_counter::report() {
35       // Note: a constant number of compares
36       //       ==> a sample standard deviation == 0.
37       for(element_counts_type::const_iterator i
38         = element_counts.begin(); i != element_counts.end(); ++i) {
39           current_ec = (*i).first;
40
41           counter_fraction avg(average());
42           double rsd(relative_std_dev());
43           double factor = avg / std::log2(double(current_ec));
44           std::cout.precision(3);
45           std::cout << (i == element_counts.begin()?"comp ":"           ");
46           std::cout << current_ec << " " << avg;
47           if(rsd != 0) std::cout << ", rsd " << rsd << "%";
48           std::cout << ",\t\t\t lg.fac " << factor << " lg n.\n";
49       }
50     }
51     template<typename PP /* = data_columns::default_post_processor*/>
52     void compare_counter::gnuplot_data(std::string description) {
53       sampler<compare_counter_base>::gnuplot_data<PP>(description+" - "+unit_name(0));
54     }
55
56     template <typename C>
57     struct counting_compare {
58       template <typename T>
59       bool operator()(T const& a, T const& b) const {
```

```
60          ++compare_counter_base::comps;
61          return C()(a,b);
62        }
63      };
64    }
65  }
66
67  #endif
```

C.5 benchmark_cpu_attention.hpp

```
1   #ifndef _CPHSTL_BENCHMARK_CPU_ATTENTION_HPP_
2   #define _CPHSTL_BENCHMARK_CPU_ATTENTION_HPP_
3   /*
4   Desc: Especially the newer multi-core cpu systems may result in
5   :     inpredictable interrupts and core shifts for current process.
6   :     Sometimes this spoils the timing methods, and an efficient
7   :     solution, on Intel cpu's, is to ask for more attention.
8   :     Also included a cross-platform (posix/msc) sleep.
9   Auth: Asger Bruun 2009-2010.
10  */
11  #ifdef _MSC_VER
12    #include <windows.h> // Sleep, QueryPerformance, Process Affinity & Priority
13  #elif defined(__GNUC__)
14    #include <time.h> // nanosleep()
15  #endif
16
17  namespace cphstl {
18    namespace benchmarking {
19  #ifdef _WIN32
20  void sleep_ms(int milliseconds) {
21    Sleep(milliseconds);
22  }
23  #else
24  void sleep_ms(long milliseconds) {
25    struct timespec tm;
26    tm.tv_sec = 0;
27    tm.tv_nsec = milliseconds * 1000000;
28    nanosleep(&tm, 0);
29  }
30  #endif
31
32  #ifdef _MSC_VER
33  void cpu_attention(bool yes) {
34    // Desc: get full attention on a single cpu core.
35    static HANDLE h_process = 0;
36    static DWORD_PTR process_affinity, system_affinity;
37
38    if(yes ^ (h_process == 0))
39      throw std::exception("mode not different from previous mode");
40    if(yes) {
41      h_process = GetCurrentProcess();
42      if(!GetProcessAffinityMask(h_process, &process_affinity, &system_affinity))
43        throw std::exception("failed to get process affinity", GetLastError());
44      //if(!SetProcessAffinityMask(h_process, 4L)) // use core #3
45      if(!SetProcessAffinityMask(h_process, 2L))
46        throw std::exception("failed to set process affinity", GetLastError());
47      if(!SetPriorityClass(GetCurrentProcess(), REALTIME_PRIORITY_CLASS))
48        throw std::exception("failed to set process priority", GetLastError());
49      if(!SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_TIME_CRITICAL))
50        throw std::exception("failed to set thread priority", GetLastError());
51    } else {
52      if(!SetProcessAffinityMask(h_process, process_affinity))
53        throw std::exception("failed to restore process affinity", GetLastError());
54      h_process = 0;
55      if(!SetPriorityClass(GetCurrentProcess(), NORMAL_PRIORITY_CLASS))
56        throw std::exception("failed to restore process priority", GetLastError());
57      if(!SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_NORMAL))
58        throw std::exception("failed to restore thread priority", GetLastError());
59    }
60  }
61  #elif defined(__GNUC__)
62  void cpu_attention(bool yes) throw(std::exception) {;}
```

```
#else
 void cpu_attention(bool yes) throw(std::exception) {;}
#endif
  }
 }
#endif
```

C.6 benchmark_custom_counter.hpp

```
#ifndef _CPHSTL_BENCHMARK_CUSTOM_COUNTER_HPP_
#define _CPHSTL_BENCHMARK_CUSTOM_COUNTER_HPP_
/*
Desc: A custom counter "array" of custom_counter<0>,.. custom_counter<N>.
      Use them to measure everyting else than time, space and compares.
      Example, You can put a ++custom_counter<0>::count; in
      a specific procedure to count its number of invokations.
Futu: Perhaps we should develop a bool counter to measure things like
      branch conditions.
Auth: Asger Bruun 2010.
*/

#include "benchmark_statistics.hpp"

namespace cphstl {
  namespace benchmarking {
    template<int index>
    struct custom_counter_base {
      typedef long long counter_type;
      static counter_type count;
      static counter_type current() { return count; }
    };
    template<int index>
    custom_counter_base<index>::counter_type
      custom_counter_base<index>::count(0);

    template<int index>
    struct custom_counter : public sampler<custom_counter_base> {
      static const unsigned int multiplicity = 1;
      static std::string unit_name(unsigned int counter_idx) {
        switch(counter_idx) {
          case 0: return  "custom";
          default: assert(0); return 0; break;
        }
      }

      custom_counter<index>(bool single_op)
        : sampler<custom_counter_base>(single_op) {}

      void start(size_type const& element_count) {
        sampler<custom_counter_base>::start(element_count);
      }
      void stop() {
        sampler<custom_counter_base>::stop();
      }

      void report();
      template<typename PP /* = data_columns::default_post_processor*/>
      void gnuplot_data(std::string description);
    };
    void custom_counter::report() {
      for(element_counts_type::const_iterator i
        = element_counts.begin(); i != element_counts.end(); ++i) {
          current_ec = (*i).first;

          counter_fraction avg(average());
          double rsd(relative_std_dev());
          double factor = avg / std::log2(double(current_ec));
          std::cout.precision(3);
          std::cout << (i == element_counts.begin()?"comp ":"           ");
          std::cout << current_ec << " " << avg;
          if(rsd != 0) std::cout << ", rsd " << rsd << "%";
          std::cout << ",\t\t\t lg.fac " << factor << " lg n.\n";
      }
```

```
65      }
66      template<typename PP /* = data_columns::default_post_processor*/>
67      template<int index>
68      void custom_counter<index>::gnuplot_data(std::string description) {
69        sampler<custom_counter_base>::gnuplot_data<PP>(description+" - "+unit_name(0));
70      }
71    }
72  }
73
74
75  #endif
```

C.7 benchmark_faster_clock_policy.hpp

```
1   #ifndef _CPHSTL_BENCHMARK_FASTER_CLOCK_POLICY_HPP_
2   #define _CPHSTL_BENCHMARK_FASTER_CLOCK_POLICY_HPP_
3   /*
4   Desc: Better clock than the std::clock.
5   Auth: Asger Bruun 2009-2010.
6   */
7
8   //#include <chrono> // nanosleep()
9
10  #ifdef _MSC_VER
11    #include <windows.h> // Sleep, QueryPerformance, Process Affinity & Priority
12  #elif defined(__GNUC__)
13    #include <sys/time.h>
14    #include <time.h>
15  #endif
16  #include "benchmark_time_counter.hpp"
17
18  namespace cphstl {
19    namespace benchmarking {
20  #ifdef _MSC_VER
21
22      struct faster_clock_policy {
23        typedef LONGLONG counter_type;
24        typedef long long hertz;
25        static counter_type current() {
26          LARGE_INTEGER t;
27          if (!QueryPerformanceCounter(&t))
28            throw std::exception("QPC() failed", GetLastError());
29          return t.QuadPart;
30        }
31        static std::string unit_name() { return "performance count"; }
32        static hertz frequency() {
33          if(frequency_ == 0) frequency_ = probe_frequency();
34          assert(frequency_ > 0);
35          return frequency_;
36        }
37        static hertz probe_frequency() {
38          LARGE_INTEGER f;
39          if(!QueryPerformanceFrequency(&f))
40            throw std::exception("QPF() failed", GetLastError());
41          return f.QuadPart;
42        }
43      private:
44        static hertz frequency_;
45      };
46      faster_clock_policy::hertz faster_clock_policy::frequency_(0);
47
48  #elif defined(__GNUC__)
49
50      struct faster_clock_policy {
51        typedef long long counter_type;
52        typedef long long hertz;
53        static counter_type current() throw(std::exception) {
54          struct timeval t;
55          gettimeofday(&t,0);
56          return counter_type(t.tv_sec * 1000000 + t.tv_usec);
57        }
58        static hertz frequency() { return 1000000; }
59      };
```

```
60
61 #else
62
63     struct faster_clock_policy : public standard_clock_policy {};
64
65 #endif
66
67   }
68 }
69 #endif
```

C.8 benchmark_main.hpp

```
1  #ifndef _CPHSTL_BENCHMARK_MAIN_HPP_
2  #define _CPHSTL_BENCHMARK_MAIN_HPP_
3
4  /*
5  Desc: Benchmark execution schedule and data generation.
6  Note: This source file is messy and could use a clean up.
7  Auth: Asger Bruun 2009-2010.
8  */
9
10 #include "ms_c_fix.hpp" // pop_cnt
11 #include "assert.h++"
12
13 #include <algorithm>
14 #include <cmath>
15 #include <iostream>
16 #include <set>
17
18 #include <cmath>
19
20 #include "benchmarking.hpp"
21 //#include "benchmark_main.hpp"
22
23 namespace cphstl {
24   namespace benchmarking {
25
26     const bool randomise_test_data = true;
27     const bool focus_pathological_data = false;
28
29     struct string_filter {
30       std::set<std::string> f;
31
32       //string_filter() : includes(true) {}
33
34       bool includes;
35
36       string_filter(std::string csv) {
37         includes = true;
38         if(csv == ".") return;
39         if(csv.length() == 0) return;
40         if(csv.at(0) == '!') {
41           includes = false;
42           csv = csv.substr(1);
43         }
44
45         std::string s;
46         std::stringstream ss(csv);
47         while(getline(ss, s, ','))
48           f.insert(s);
49       }
50
51       bool pass(std::string name) const {
52         if(includes) return (f.empty() || (f.find(name) != f.end()));
53         else return (f.empty() || (f.find(name) == f.end()));
54       }
55     };
56
57     struct benchmark_base {
58       typedef std::size_t size_type;
59
60       virtual std::string name() = 0;
```

```
61        virtual std::string description() { return name(); }
62
63        virtual void prepare(size_type const& elements
64          , size_type const& repeats) = 0;
65
66        virtual void run(size_type const& element_count
67          , size_type const& repetition) = 0;
68
69        virtual void finish() = 0;
70        virtual void report() = 0;
71        virtual void gnuplot_data() = 0;
72        virtual ~benchmark_base() {}
73      };
74
75        template <typename Counter /* = combi_counter*/>
76        struct benchmark_with_counter : public benchmark_base {
77          typedef benchmark_base::size_type size_type;
78          Counter counter;
79
80          static std::string unit_name() { return Counter::unit_name(); }
81
82          benchmark_with_counter(bool single_op = false)
83            : counter(single_op) {}
84
85          void prepare(size_type const& elements, size_type const& repeats) {
86            (*this).counter.prepare(elements, repeats);
87          }
88          void finish() {
89            (*this).counter.finish();
90          }
91          void report() {
92            if(name().length()>5) std::cout << "\n" << name().substr(0,4)
93              << ".␣";
94            else {
95              std::string::size_type l(5 - name().length());
96              std::cout << "\n" << name() << std::string('␣', (char) l) << "␣";
97            }
98            (*this).counter.report();
99          }
100
101         void gnuplot_data() {
102           (*this).counter.template gnuplot_data<data_columns::default_post_processor>(name())
                ;
103         }
104       };
105
106
107     struct benchmark_suite_base : public benchmark_base {
108       typedef std::vector<benchmark_base*>::iterator iterator;
109       typedef std::vector<benchmark_base*> methods_list;
110       methods_list benchmarks_;
111
112       virtual void type_report() = 0;
113       virtual bool focus(size_type const& elements) = 0;
114     };
115
116     template <typename Counter /* = combi_counter*/>
117     struct benchmark_suite : public benchmark_suite_base {
118     private:
119
120       benchmark_suite<Counter>(benchmark_suite const&);
121       benchmark_suite& operator = (benchmark_suite const&);
122
123     public:
124
125       typedef benchmark_base::size_type size_type;
126       typedef benchmark_with_counter<Counter> benchmark;
127
128       std::string friendly_name;
129       string_filter & benchmark_filter;
130
131       benchmark_suite(std::string friendly_name, string_filter & benchmarks)
132         : friendly_name(friendly_name), benchmark_filter(benchmarks) {}
133
```

```
134         ~benchmark_suite() {
135           while(!benchmarks_.empty()) {
136             delete(benchmarks_.back()); benchmarks_.pop_back();
137           }
138         }
139
140         template <typename Benchmark>
141         void registrate(std::string name) {
142           if(benchmark_filter.pass(name)) {
143             benchmarks_.push_back(new Benchmark());
144           }
145         }
146
147         std::string name() {
148           std::string s(friendly_name);
149           return s;
150         }
151
152         void prepare(size_type const& elements, size_type const& repeats) {
153           for(iterator i = benchmarks_.begin(); i != benchmarks_.end(); ++i)
154             (*(*i)).prepare(elements,repeats);
155         }
156
157         void finish() {
158           for(iterator i = benchmarks_.begin(); i != benchmarks_.end(); ++i)
159             (*(*i)).finish();
160         }
161
162
163         void run(size_type const& element_count, size_type const& repetition) {
164           for(iterator i = benchmarks_.begin(); i != benchmarks_.end(); ++i) {
165             (*(*i)).run(element_count, repetition);
166           }
167         }
168
169         void report() {
170           std::cout << "\n" << friendly_name;
171           for(iterator i = benchmarks_.begin(); i != benchmarks_.end(); ++i)
172             (*(*i)).report();
173         }
174
175         void gnuplot_data() {
176           std::cout << "\n#␣" << friendly_name;
177           for(iterator i = benchmarks_.begin(); i != benchmarks_.end(); ++i)
178             (*(*i)).gnuplot_data();
179         }
180       };
181
182       struct default_impl_translation {
183         static std::string translate(std::string const& desc) {
184           return str_replace(desc,"_","-");
185         }
186       };
187
188       template <typename Counter /* = combi_counter*/
189         , template <typename, typename> class Driver
190         , typename T = default_impl_translation
191       >
192       struct benchmark_main {
193       private:
194
195         typedef std::vector<benchmark_suite_base*>::iterator iterator;
196         typedef std::vector<benchmark_suite_base*>::reverse_iterator reverse_iterator;
197
198         std::vector<benchmark_suite_base*> suites_;
199
200         string_filter methods_filter;
201         string_filter benchmark_filter;
202
203       public:
204
205         benchmark_main(std::string method_csv, std::string impl_csv)
206           : methods_filter(method_csv), benchmark_filter(impl_csv)
207         {}
```

```
208
209         template <typename Impl>
210         void registrate(std::string impl_name) {
211           if(benchmark_filter.pass(impl_name)) {
212             const int ws(sizeof(void*));
213
214             typedef typename Impl::realizator_type R;
215
216             std::cout << "# "
217               << (has_member_meld<R>::value?"M":"_")
218               << (has_member_clear<R>::value?"C":"_")
219               << (has_member_pop<R>::value?"P":"_")
220               << (has_member_root_list_type<typename R::encapsulator_type>::value?"R":"_")
221               << " " << impl_name;
222             std::cout << " - valuetype " << (1.0*sizeof(typename Impl::value_type)/ws) << "
                  words";
223             std::cout << ", encapsulator " << (1.0*sizeof(typename Impl::encapsulator_type)/ws)
                   << " words";
224             std::cout << ", realizator " << (1.0*sizeof(typename Impl::realizator_type)/ws) <<
                  " words.";
225             std::cout << std::endl;
226
227             suites_.push_back(new Driver<Impl, Counter>(impl_name, methods_filter));
228           }
229         }
230
231       private:
232
233         void prepare_size(int s, int r) {
234           for(iterator i = suites_.begin(); i != suites_.end(); ++i)
235             (*(*i)).prepare(s, r);
236         }
237         void run_size(int s, int r) {
238           std::cerr << "progress: size " << s << ", repetitions " << r;
239           stop_watch sw(false);
240           sw.start(1);
241           for(int rep(0); rep != r; ++rep) {
242             for(iterator i = suites_.begin(); i != suites_.end(); ++i) {
243               (*(*i)).run(s, rep);
244             }
245           }
246           //}
247           sw.stop();
248           std::cerr << ", time " << sw.average_time() << " s.\n";
249         }
250
251         void write_head() {
252           std::cout << "# implementations: ";
253           for(iterator i = suites_.begin(); i != suites_.end(); ++i) {
254             if(i != suites_.begin()) std::cout << ",";
255             std::cout << (*(*i)).description();
256           }
257           assert(!suites_.empty());
258           std::cout << "\n# methods: ";
259           benchmark_suite_base* bs(*suites_.begin());
260           for(benchmark_suite_base::iterator i((*bs).benchmarks_.begin());
261             i != (*bs).benchmarks_.end(); ++i)
262           {
263             if(i != (*bs).benchmarks_.begin()) std::cout << ",";
264             std::cout << (*(*i)).name();
265           }
266           std::cout << "\n";
267         }
268
269       public:
270
271         void run(int max_exp, int c, int min_rep, bool test) {
272           std::cout << "\n# benchmark data\n";
273           write_head();
274           std::cout << "\n# max size: 2^" << max_exp << " (" << (1 << max_exp)
275             << ") with " << c << " intervals between every power of 2.";
276
277           std::cout << "\n\n# standard clock ";
278           time_counter<standard_clock_policy>::report_clock_specs();
```

```
279         std::cout << "# faster clock ";
280         time_counter<faster_clock_policy >::report_clock_specs();
281         std::cout << "# stop watch " ;
282         stop_watch::report_clock_specs();
283         std::cout << "# tsc ";
284         time_counter<tsc_clock_policy >::report_clock_specs();
285         std::cout << "#\n";
286 #ifdef _MSC_VER
287         std::cout << "# cpu_has_pop_cnt(): " << cphstl::cpu_has_pop_cnt() << ".\n";
288         std::cout << "# cpu_has_lz_cnt(): " << cphstl::cpu_has_lz_cnt() << ".\n";
289         std::cout << "\n";
290 #endif
291         typedef std::map<const long, long> exe_plan_type;
292         exe_plan_type exe_plan;
293
294         if(test)
295           for(int s = 1; s != (1 << max_exp); ++s) run_size(s, 1); // exe_plan[s] = 1;
296         else {
297           //  // Desc: uneven distribution: 2^k + j*2^(1+k-log(inter)) - 1, j = 0..inter
                -1.
298           //  //       the least significant bits are all zeros (and if size-1 then ones).
299           //  // Ex.2: 2^21-1, 2^21, 2^21+2^20-1, 2^21+2^20,..
300           //  // Ex.4: 2^21-1, 2^21, 2^21+2^19-1, 2^21+2^19, 2^21+2*2^19-1, 2^21+2*2^19,..
301           {
302             // Desc: even distribution: 2^(k/inter).
303             assert(!suites_.empty());
304             benchmark_suite_base* bs(*suites_.begin());
305             int m(max_exp * c), end(m+1);
306             int last(0);
307             for(int k(c); k != end; ++k) {
308               int s = int(pow(double(2), double(k)/double(c)));
309               int r = int(pow(double(2), double(m+c-k)/c));
310               r >>= 6;
311               r >>= 2;
312               if(r < min_rep) r = min_rep;
313
314               if(s != last) {
315                 for(int s2(last+1); s2<= s; ++s2)
316                   if(focus_pathological_data && (*bs).focus(s2)) exe_plan[s2] = r;
317                 exe_plan[s] = r; last = s;
318               }
319             }
320           }
321
322           std::cout << "# execution plan: size x repetitions.";
323           int brk(0);
324           for(exe_plan_type::const_reverse_iterator j
325             = exe_plan.rbegin(); j != exe_plan.rend(); ++j) {
326             std::cout << ((brk == 0)?"\n#\t":"\t");
327             std::cout << (*j).first << " x "
328               << (*j).second << " = " << ((*j).first * (*j).second) << ".";
329             brk = (brk + 1) % 4;
330           }
331           std::cout << "\n";
332
333 #ifdef NDEBUG
334           benchmarking::cpu_attention(true);
335 #endif
336           stop_watch total(false);
337           total.start(1);
338           for(exe_plan_type::const_reverse_iterator j
339             = exe_plan.rbegin(); j != exe_plan.rend(); ++j) {
340             prepare_size((*j).first, (*j).second);
341
342             //benchmarking::cpu_attention(true);
343             run_size((*j).first, (*j).second);
344             //benchmarking::cpu_attention(false);
345             //sleep_ms(10);
346           }
347 #ifdef NDEBUG
348           benchmarking::cpu_attention(false);
349 #endif
350           total.stop();
351           int total_seconds(total.average_time());
```

```
352             int seconds(total_seconds % 60);
353             int hours(total_seconds / 60);
354             std::cout << "\n# total execution time " << hours << " m :" << seconds << " s.\n";
355           }
356           for(iterator i = suites_.begin(); i != suites_.end(); ++i)
357             (*(*i)).finish();
358         }
359
360         void report() {
361           std::cout << "\n";
362           for(iterator i = suites_.begin(); i != suites_.end(); ++i)
363             (*(*i)).report();
364         }
365
366         void type_report() {
367           std::cout << "\n";
368           for(iterator i = suites_.begin(); i != suites_.end(); ++i)
369             (*(*i)).type_report();
370         }
371
372         void gnuplot_data() {
373           benchmark_suite_base* bs(*suites_.begin());
374           if((*bs).benchmarks_.size() == 1)
375             std::cout << "\n# " << (*(*(*bs).benchmarks_.begin())).name() << ".dat";
376           else
377             std::cout << "\n# pq.dat";
378           std::cout << " (test data " << (randomise_test_data?"random":"repeated") << ")";
379           for(iterator i = suites_.begin()
380             ; i != suites_.end(); ++i) (*(*i)).gnuplot_data();
381             std::cout << "\nend\n";
382         }
383
384         void gnuplot_script(int max_exp) {
385           std::cout << "\n# gnuplot script\n";
386           write_head();
387
388           bool plot = true;
389           bool stat = true;
390           bool minimum = true;
391           bool smooth = true;
392           bool gen[] = {plot, stat, minimum, smooth};
393
394           bool title = true;
395
396           bool key = true;
397           bool ext_palette = false;
398           bool use_style_line = false; // broken
399
400           bool lines = true;
401           bool points = true;
402           bool custom_points = true; // select my symbols
403           int custom_pts[] = {4,6,8,10,12};
404           int custom_pts_len = sizeof(custom_pts)/sizeof(custom_pts[0]);
405           bool var_points = focus_pathological_data; // variable sized points
406
407           std::string ext = "eps";
408           //std::string ext = "ps";
409
410           // test data column indices.
411           const int pct5 = data_columns::pct5+2;  // 2
412           const int q1 = data_columns::Q1+2;      // 3
413           const int median = data_columns::median+2;// 4
414           const int q3 = data_columns::Q3+2;      // 5
415           const int pct95 = data_columns::pct95+2;// 6
416           const int min_v = data_columns::min_v+2;
417           //const int max_v = data_columns::max_v+2;
418           const int s = data_columns::user_def+2;
419
420
421           std::vector<float> col(suites_.size());
422           for(unsigned int j = 0; j != suites_.size(); ++j)
423             col[j] = (float(j)/float(suites_.size()-1));
424             //col[j] = (float(j)/float(suites_.size()));
425
```

```
426         assert(!suites_.empty());
427         for(int bw(0); bw !=  sizeof(gen)/sizeof(gen[0]); ++bw) {
428           benchmark_suite_base* bs(*suites_.begin());
429           benchmark_suite_base::methods_list & m((*bs).benchmarks_);
430
431           if(gen[bw]) for(unsigned int i = 0; i != m.size(); ++i) {
432             if(ext == "eps")
433               std::cout << "\nset terminal postscript eps color dashed enhanced font \"Times\"
                      14";
434             else if(ext == "ps")
435               std::cout << "\nset terminal postscript landscape color dashed enhanced font \"
                      Times New Roman\" 14";
436             else if(ext == "svg")
437               std::cout << "\nset terminal svg dashed enhanced size 1440,1024";
438             else if(ext == "png")
439               std::cout << "\nset terminal png transparent nocrop enhanced size 1440,1024";
440             else if(ext == "pdf")
441               std::cout << "\nset terminal pdf";
442             else { std::cerr << "bad gnuplot output file type: " << ext; std::abort(); }
443
444             std::cout << "\n" << (key?"":"# ") << "set key top left samplen 8";
445             std::cout << "\n# set key below 1.25";
446             std::cout << "\n" << (key?"# ":"") << "unset key";
447
448             if(title) std::cout << "\nset xlabel 'elements'";
449             std::cout << "\nset logscale x 2"
450               << "\nset autoscale"
451               << "\nset yrange [ 0.00000 : ]"
452               << "\nset xrange [ 1.00000 : " << (1 << (max_exp+1)) << " ]"
453               << "\nset format x \"2^{%L}\""
454               << "\nset grid"
455               << "\n";
456             /*if(!var_points)*/
457               //std::cout << "set pointsize 0.5\n";
458               //std::cout << "set pointsize 0.4\n";
459               std::cout << "set pointsize 0.5\n";
460
461
462             if(ext_palette) {
463               std::cout << "\nset palette model RGB";
464               std::cout << "\nset palette file 'pq.pal'";
465             } else
466               std::cout << "\nset palette defined ( 0 'dark-green', 1 'green', 2 'blue', 3 '
                      dark-blue', 4 'dark-red', 5 'orange' )";
467               //std::cout << "\nset palette defined ( 0 'dark-green', 1 'green', 2 'blue',
                        3 'dark-blue', 4 'red', 5 'dark-orange' )";
468               //std::cout << "\nset palette defined ( 0 'green', 1 'blue', 2 'red', 3 '
                      orange' )";
469               //std::cout << "\nset palette defined ( 0 'red', 1 'green', 2 'blue', 3 '
                      magenta', 4 'coral' )";
470               //std::cout << "\nset palette defined ( 0 'dark-green', 1 'red', 3 'blue' )
                      ";
471               //std::cout << "\nset palette defined ( 0 'dark-green', 1 'blue', 2 'blue-
                      violet', 3 'dark-red', 4 'dark-orange' )";
472               //std::cout << "\nset palette defined ( 0 'dark-green', 1 'dark-cyan', 2 '
                      blue', 3 '#8A2BE2', 4 'dark-red', 5 'dark-orange' )";
473               //std::cout << "\nset palette defined ( 0 'dark-green', 1 'blue', 2 '#8A2BE2
                      ', 3 'dark-red', 4 'dark-orange' )";
474
475 //set palette defined (0 "dark-green", 1 "green", 1 "yellow", \
476 2 "dark-yellow", 2 "red", 3 "dark-red" )
477
478             if(use_style_line) {
479               for(unsigned int j = 0; j != suites_.size(); ++j) {
480                 std::cout << "\nset style line " << (j+1) << " lt " << (j % 8 + 2);
481                 std::cout << " lc pal frac " << col[j];
482               }
483             }
484             std::cout << "\nunset colorbox";
485             std::cout << "\n";
486
487             for(unsigned int k = 0; k != Counter::multiplicity; ++k) {
488               std::cout << "\nset output '"
489                 << (bw == 1?"stat_":bw == 2?"min_":bw == 3?"smooth_":"plot_")
```

```
490                 << str_replace(Counter::unit_name(k)," ","_") << "_"
491                 << (*m[i]).name() << "." << ext << "'";
492
493               if(title) std::cout << "\nset title \""
494                 << str_replace((*m[i]).description(),"_"," ") << "\""
495                 << "\nset ylabel '" << Counter::unit_name(k)
496                 << "'";
497
498
499               //if(bw == 3) std::cout << "\nsw(x,S) = 1000/(x*x*S)";
500               if(bw == 3) std::cout << "\nsw(x,S) = (100/(S*x*x*x))";
501               //if(bw == 3) std::cout << "\nsw(x,S) = (10/(S*(1.00002**x)))";
502               //if(bw == 3) std::cout << "\nsw(x,S) = (0.00001)";
503               for(unsigned int j = 0; j != suites_.size(); ++j) {
504                 if(j == 0) {
505                   std::cout << "\nplot '";
506                   benchmark_suite_base* bs(*suites_.begin());
507                   if((*bs).benchmarks_.size() == 1)
508                     std::cout << (*(*(*bs).benchmarks_.begin())).name() << ".dat";
509                   else std::cout << "pq.dat";
510                   std::cout << "'";
511                 } else std::cout << ", ''";
512
513                 typename Counter::counter_type idx(k
514                   + Counter::multiplicity * (i + j * m.size()));
515
516                 std::cout << " using 1:" << (bw != 2?median:min_v);
517                 if(var_points && points && bw == 0) std::cout << ":" << s;
518                 if(bw == 3) std::cout << ":(sw($1,1)) ";
519
520                 //if(bw == 3) std::cout << ":(1e-4) ";
521                 //if(bw == 3) std::cout << ":(1.0) ";
522                 std::cout << " index " << idx;
523                 if(bw == 3) std::cout << " smooth acsplines ";
524                 std::cout << " with ";
525
526                 if(points && lines) std::cout << "linespoints ";
527                 else if(points) std::cout << "points ";
528                 else if(lines) std::cout << "lines ";
529
530                 if(points && custom_points) std::cout << " pt " << custom_pts[(j+1) %
                      custom_pts_len];
531                 else if(points) std::cout << " pt " << (j+1);
532
533                 //if(lines || bw != 0)
534                 {
535                   if(use_style_line) std::cout << " ls " << (j+1);
536                   else std::cout << " lt " << (j % 8 + 2) << " lc pal frac "
537                     << col[j];
538                 }
539                 if(var_points && points && bw == 0) std::cout << " pointsize variable ";
540
541                 std::cout << " title \""
542                   << str_replace((*suites_[j]).description(),"_","-")
543                   << "\"";
544
545                 if(bw == 1) std::cout << ", '' using 1:" << q1 << ":"
546                   << pct5 << ":" << pct95 << ":" << q3 << " index " << idx
547                   << " with candlesticks lt 1 lc pal frac "
548                   << col[j]
549                   << " notitle whiskerbars";
550                 //  << " w yerr linecolor " << (j+1) << "";
551               }
552             }
553           }
554         }
555         if(true || !key) {
556           std::cout << "\nset output '"
557             << "legends." << ext << "'"
558             << "\nunset title"
559             //<< "\nset key ins vert\nset key left top"
560             << "\nset key below"
561             << "\nunset xlabel"
562             << "\nunset ylabel"
```

```
563             << "\nunset xrange; unset yrange"
564             << "\nunset border; unset xtics; unset ytics; unset ztics";
565
566           for(unsigned int j = 0; j != suites_.size(); ++j) {
567             std::cout << (j == 0?"\nplot 'dummy.dat':",','") << " using 1:" << median << "
                  with ";
568             if(points && lines) std::cout << "linespoints ";
569             else if(points) std::cout << "points ";
570             else if(lines) std::cout << "lines ";
571
572             if(points && custom_points) std::cout << " pt " << custom_pts[(j+1) %
                  custom_pts_len];
573             else if(points) std::cout << " pt " << (j+1);
574
575             //if(lines || bw == 1)
576             {
577               if(use_style_line) std::cout << " ls " << (j+1);
578               else std::cout << " lt " << (j % 8 + 2) << " lc pal frac "
579                 << col[j];
580             }
581
582             std::cout << " lw 3";
583             std::cout << " title \""
584               << T::translate((*suites_[j]).description()) << "\"";
585           }
586         }
587       }
588
589       ~benchmark_main() {
590         while(!suites_.empty())
591         { delete(suites_.back()); suites_.pop_back(); }
592       }
593     };
594   } // namespace benchmarking
595 } // namespace cphstl
596 #endif
```

C.9 benchmark_profiling_counter.hpp

```
1  #ifndef _CPHSTL_BENCHMARK_PROFILING_COUNTER_HPP_
2  #define _CPHSTL_BENCHMARK_PROFILING_COUNTER_HPP_
3  /*
4  Desc: A hook in to external Valgrind profiling tool, guides the
5        profiler to measure only the parts between counter start
6        and stop. This speeds up your external profiling because
7        irrelevant parts are skipped and reduces the loads of data
8        collected to be analysed by You later.
9  Auth: Asger Bruun 2010.
10 Ref:  http://valgrind.org/docs/manual/cl-manual.html
11
12 */
13
14 #if defined(__GNUC__)
15 #include </usr/include/valgrind/callgrind.h>
16 #else
17   #define CALLGRIND_START_INSTRUMENTATION
18   #define CALLGRIND_STOP_INSTRUMENTATION
19 #endif
20
21 #include "benchmark_statistics.hpp"
22
23 namespace cphstl {
24   namespace benchmarking {
25     struct profiling_counter_base {
26       typedef long long counter_type;
27       static counter_type profilings;
28       static counter_type current() { return profilings; }
29     };
30     profiling_counter_base::counter_type profiling_counter_base::profilings(0);
31
32     struct profiling_counter : public sampler<profiling_counter_base> {
33       static const unsigned int multiplicity = 1;
34       static std::string unit_name(unsigned int counter_idx) {
```

```
35          switch(counter_idx) {
36            case 0: return  "profilings";
37            default: assert(0); return 0; break;
38          }
39        }
40
41        profiling_counter(bool single_op) : sampler<profiling_counter_base>(single_op) {}
42
43        void start(size_type const& element_count) {
44          sampler<profiling_counter_base>::start(element_count);
45          CALLGRIND_START_INSTRUMENTATION;
46        }
47        void stop() {
48          CALLGRIND_STOP_INSTRUMENTATION;
49          ++profilings;
50          sampler<profiling_counter_base>::stop();
51        }
52
53        void report();
54        template<typename PP /* = data_columns::default_post_processor*/>
55        void gnuplot_data(std::string description);
56      };
57      void profiling_counter::report() {
58        // Note: a constant number of compares
59        //       ==> a sample standard deviation == 0.
60        for(element_counts_type::const_iterator i
61          = element_counts.begin(); i != element_counts.end(); ++i) {
62            current_ec = (*i).first;
63
64            counter_fraction avg(average());
65            double rsd(relative_std_dev());
66            double factor = avg / std::log2(double(current_ec));
67            std::cout.precision(3);
68            std::cout << (i == element_counts.begin()?"comp ":"           ");
69            std::cout << current_ec << " " << avg;
70            if(rsd != 0) std::cout << ", rsd " << rsd << "%";
71            std::cout << ",\t\t\t lg.fac " << factor << " lg n.\n";
72        }
73      }
74      template<typename PP /* = data_columns::default_post_processor*/>
75      void profiling_counter::gnuplot_data(std::string description) {
76        sampler<profiling_counter_base>::gnuplot_data<PP>(description+" - "+unit_name(0));
77      }
78    }
79  }
80
81
82  #endif
```

C.10 benchmark_statistics.hpp

```
1   #ifndef _CPHSTL_BENCHMARK_STATISTICS_HPP_
2   #define _CPHSTL_BENCHMARK_STATISTICS_HPP_
3   /*
4   Desc: Repeated measurements of resource consumption.
5   :     Is an attempt to simplify measurements.
6   :     Includes some basic statistics.
7   Auth: Asger Bruun 2009-2010.
8   Impl: sampler - the base class for performance counters.
9   Note: Bessel error correction for small sample sizes is used.
10  */
11  #include "assert.h++"
12  #include <vector>
13  #include <map>
14  #include <numeric> // accumulate
15  #include <sstream> // ostringstream
16  //#include <iostream>
17  #include <algorithm> // copy
18  #include <cmath> // sqrt
19
20
21  namespace cphstl {
22    namespace benchmarking {
```

```
23      /* interface "sampler counter base" {
24      *    typedef <user defined> counter_type;
25      *    static counter_type current();
26      *  }; */
27
28      std::string to_str(long const& n) {
29        std::ostringstream s;
30        s << n;
31        return s.str();
32      }
33
34      namespace data_columns {
35        enum data_column_index {
36          pct5, Q1, median, Q3, pct95
37          , min_v, max_v
38          , avg
39          //, sample_std_dev, rel_std_dev
40          , user_def
41          , bowley, repeats
42          , column_count
43        };
44
45        void write_header() {
46          const char headers[12][10]
47          = {"count","pct5","Q1","median","Q3","pct95","min","max","avg","user_def","bowley","
                repeats"};
48          const int headers_len = sizeof(headers)/sizeof(headers[0]);
49          assert((column_count+1) == headers_len);
50          std::cout << "\n#␣";
51          for(int i(0); i != headers_len; ++i) {
52            if(i != 0) std::cout << "\t";
53            std::cout << headers[i];
54          }
55          std::cout << "\n";
56        }
57
58        typedef std::size_t size_type;
59        typedef std::vector<double> row_type;
60
61        void write_data(size_type const& count, row_type & row) {
62          assert(column_count == row.size());
63          std::cout << count << "\t";
64          copy (row.begin(), row.end()
65             , std::ostream_iterator<double>(std::cout,"\t"));
66          std::cout << "\n";
67        }
68
69        struct default_post_processor {
70          static void process(size_type const& /*count*/, row_type & row) {
71            assert(column_count == row.size());
72            row[user_def] = 1;
73          }
74        };
75      }
76
77
78      template <typename B> // B = Counter Base.
79      struct sampler : public B {
80        // Desc: sampling and basic statistics.
81        // Ref:  en.wikipedia.org/wiki/Standard_error_(statistics)
82        //       en.wikipedia.org/wiki/Standard_deviation
83        typedef typename B::counter_type counter_type;
84        typedef long double counter_fraction;
85        typedef std::size_t size_type;
86        typedef std::vector<counter_type> samples_type;
87        typedef std::map<const size_type, samples_type> element_counts_type;
88
89        bool single_op;
90
91        sampler(bool single_op/* = false*/) : single_op(single_op)
92          , current_ec(-1), start_c(-1) {}
93
94        ~sampler() {
95          for(typename element_counts_type::iterator i
```

```
96            = element_counts.begin(); i != element_counts.end(); ++i) {
97              (*i).second.reserve(0);
98              (*i).second.clear();
99          }
100         element_counts.clear();
101       }
102
103       void prepare(size_type const& elements, size_type const& repeats) {
104         element_counts[elements].reserve(repeats);
105       }
106       void start(size_type const& element_count);
107       void stop();
108       void finish() {
109         // Proposal: generate test results here.
110       }
111
112       size_type size(); // number of samples
113       counter_fraction average();
114       counter_fraction sample_std_dev(); // sample standard deviation.
115       double relative_std_dev(); // relative standard error.
116
117       std::vector<double> box_and_whispers();
118       std::vector<double> quartiles(); // {min, Q1, Q2, Q3, max}
119
120       std::vector<double> quantile(int numerator, int denominator);
121
122       samples_type& samples() {
123         assert(current_ec != -1);
124         return element_counts[current_ec];
125       }
126
127       counter_type current_ec;
128
129       static const unsigned int multiplicity = 1;
130
131
132       template<typename PP /* = data_columns::default_post_processor*/>
133       void gnuplot_data(std::string description) {
134         if(description.length() != 0)
135           std::cout << "\n#␣" << description;
136         data_columns::write_header();
137         for(typename element_counts_type::const_iterator i
138           = element_counts.begin(); i != element_counts.end(); ++i)
139         {
140             current_ec = (*i).first;
141             std::vector<double> row(box_and_whispers());
142             row.reserve(data_columns::column_count);
143             row.push_back(average());
144             //row.push_back(relative_std_dev()); row.push_back(sample_std_dev());
145             row.push_back(1); // user_def
146
147             double IQR = row[data_columns::Q3]
148             - row[data_columns::Q1]; // interquartile range
149             double skewness(0);
150             if(IQR != 0) skewness = (row[data_columns::Q1]+row[data_columns::Q3]
151             - 2*row[data_columns::median]) / IQR;
152             row.push_back(skewness); // bowley
153             row.push_back(samples().size()); // repetition count
154
155             assert(data_columns::column_count == row.size());
156             PP::process(current_ec, row);
157             assert(data_columns::column_count == row.size());
158
159             data_columns::write_data(current_ec, row);
160         }
161         std::cout << "\n";
162       }
163
164       //void gnuplot_data(std::string description) {
165       //  gnuplot_data<data_columns::default_post_processor>(description);
166       //}
167
168
169     protected:
```

```
170         element_counts_type element_counts;
171         counter_type start_c;
172
173       private:
174
175         counter_fraction per_op(const counter_fraction& v) {
176           // Desc: if not single operation then assume
177           // current_ec number of operations and divide.
178           return v / counter_fraction(single_op?1:current_ec);
179         }
180       };
181       template <typename B>
182       void sampler<B>::start(size_type const& element_count) {
183         assert(start_c == -1); start_c = B::current();
184         assert(element_count >= 0); current_ec = element_count;
185       }
186       template <typename B>
187       void sampler<B>::stop() {
188         assert(start_c != -1); counter_type now = B::current();
189         samples().push_back(now - start_c);
190         start_c = -1;
191       }
192       template <typename B>
193       typename sampler<B>::size_type sampler<B>::size() {
194         return samples().size();
195       }
196       template <typename B>
197       typename sampler<B>::counter_fraction sampler<B>::average() {
198         return per_op(counter_fraction(accumulate(samples().begin(), samples().end(),
                counter_type(0)))
199           / counter_fraction(samples().size()));
200       }
201       template <typename B>
202       typename sampler<B>::counter_fraction sampler<B>::sample_std_dev() {
203         // Desc: sample standard deviation, with Bessel's correction.
204         // Warn: to use std_dev one needs to know the nature of the distribution.
205         counter_fraction avg(average());
206         std::vector<counter_fraction> residuals(samples().begin(), samples().end());
207         transform(residuals.begin(), residuals.end()
208           , residuals.begin(), bind2nd(std::minus<counter_fraction>(), avg));
209         counter_fraction rip = inner_product(residuals.begin(), residuals.end()
210           , residuals.begin(), counter_fraction(0));
211         return per_op(std::sqrt(rip / counter_fraction(samples().size() - 1)));
212       }
213       template <typename B>
214       double sampler<B>::relative_std_dev() { // relative standard error
215         if(average() == 0) return -1; // (perhaps better throw?)
216         return double(sample_std_dev()) * 100 / double(average());
217       }
218
219       template <typename B>
220       std::vector<double> sampler<B>::box_and_whispers() {
221         // ret: (pct10, Q1, median, Q3, pct90, min, max).
222         const int d = 100; // percentile denominator
223         const int quantiles[5] = {10,25,50,75,90};
224
225         std::vector<double> ret(5+2);
226         sort(samples().begin(), samples().end());
227         //samples_type& s(samples());
228         size_type last(samples().size()-1);
229         for(int i = 0; i != 5; ++i) {
230           assert(quantiles[i]>= 0 && quantiles[i]<= d);
231           size_type p_q(last*quantiles[i]);
232           size_type p(p_q / d);
233           int m(p_q % d);
234           if(m == 0) ret[i] = per_op(samples()[p]);
235           else { // interpolate
236             // simpler: ret[i] = per_op(double(samples()[p])/2 + double(samples()[p+1])/2);
237             ret[i] = per_op(double(samples()[p+1])*m/d
238               + double(samples()[p])*(d-m)/d);
239           }
240           if(i != 0) { assert(ret[i]>= ret[i-1]); }
241         }
242         ret[5] = per_op(samples()[0]);
```

```
243         ret[6] = per_op(samples()[last]);
244         return ret;
245       }
246
247       template <typename B>
248       std::vector<double> sampler<B>::quartiles() {
249         const int quantiles = 4; // (4  => quartiles).
250         std::vector<double> ret(quantiles+1);
251         sort(samples().begin(), samples().end());
252         size_type last(samples().size()-1);
253         ret[0] = samples()[0];
254         for(int i = 1; i != quantiles; ++i) {
255           size_type p_q(last*i);
256           size_type p(p_q / quantiles);
257           int m(p_q % quantiles);
258           if(m == 0) ret[i] = per_op(samples()[p]);
259           else {
260             ret[i] = per_op(double(samples()[p])*m/quantiles
261               + double(samples()[p+1])*(quantiles-m)/quantiles);
262           }
263         }
264         ret[quantiles] = samples()[last];
265         return ret;
266       }
267     }
268   }
269
270   #endif
```

C.11 benchmark_time_counter.hpp

```
1   #ifndef _CPHSTL_BENCHMARK_TIME_COUNTER_HPP_
2   #define _CPHSTL_BENCHMARK_TIME_COUNTER_HPP_
3   /*
4   Desc: A std::clock based time counter.
5   Auth: Asger Bruun 2009-2010.
6   */
7   #include <ctime>
8   #include <stdexcept>
9   #include <algorithm> // std::min
10  #include "benchmark_statistics.hpp"
11  #include "benchmark_tsc_clock_policy.hpp"
12
13  namespace cphstl {
14    namespace benchmarking {
15      struct standard_clock_policy {
16        typedef std::clock_t counter_type;
17        typedef long long hertz;
18        static counter_type current() {
19          counter_type t(std::clock());
20          if (t == std::clock_t(-1)) {
21            throw std::runtime_error("No clock");
22          }
23          return t;
24        }
25        static hertz frequency() { return CLOCKS_PER_SEC; }
26        static std::string unit_name() { return "1/"+to_str(hertz())+" seconds"; }
27      };
28
29      template <typename P = tsc_clock_policy>
30      //template <typename P = faster_clock_policy>
31      //template <typename P = standard_clock_policy>
32      struct time_counter : public sampler<P> {
33        // Desc: platform independant arbitrary time measurement.
34        typedef typename P::counter_type counter_type;
35        typedef typename P::hertz hertz;
36
37        static const unsigned int multiplicity = 1;
38        static std::string unit_name(unsigned int counter_idx) {
39          switch(counter_idx) {
40            case 0: return  P::unit_name();
41            default: assert(0); return 0; break;
42          }
```

```
43        }
44
45        time_counter(bool single_op) : sampler<P>(single_op) {}
46
47        void report() {
48          for(typename sampler<P>::element_counts_type::const_iterator i
49            = (*this).element_counts.begin(); i != (*this).element_counts.end(); ++i) {
50              (*this).current_ec = (*i).first;
51              typename sampler<P>::counter_fraction avg((*this).average());
52              double rsd((*this).relative_std_dev());
53              double factor = avg / std::log2(double((*this).current_ec));
54              std::cout.precision(3);
55              std::cout << (i == (*this).element_counts.begin()?"time ":"          ");
56              std::cout << (*this).current_ec << " " << avg;
57              std::cout << ", rsd " << rsd << "%";
58              std::cout << ", \t lg.fac " << factor << " lg n.\n";
59          }
60        }
61        template<typename PP /* = data_columns::default_post_processor*/>
62        void gnuplot_data(std::string description) {
63          sampler<P>::template gnuplot_data<PP>(description+" - "+unit_name(0));
64        }
65        static void report_clock_specs() {
66          hertz f(sampler<P>::frequency());
67          counter_type g(granularity());
68          std::cout << "frequency: "  << f << " hz, granularity: "
69            << g << " hz (" << double(g)/f << " s).\n";
70        }
71        static counter_type granularity() {
72          return std::min(probe_granularity(),probe_granularity());
73        }
74
75      private:
76        static counter_type probe_granularity(){
77          counter_type t1(sampler<P>::current());
78          counter_type t2(sampler<P>::current());
79          while(t1 == t2) t2 = sampler<P>::current();
80          return (t2-t1);
81        }
82      };
83    }
84  }
85  #endif
```

C.12 benchmark_tsc_clock_policy.hpp

```
1   #ifndef _CPHSTL_BENCHMARK_TSC_CLOCK_POLICY_HPP_
2   #define _CPHSTL_BENCHMARK_TSC_CLOCK_POLICY_HPP_
3   /*
4   Desc: This module counts clock cycles.
5   Auth: Asger Bruun 2009-2010.
6   */
7   #include <algorithm> // std::max
8   #include "benchmark_faster_clock_policy.hpp" // fallback if no platform and compiler
        support.
9
10  namespace cphstl {
11    namespace benchmarking {
12      struct tsc_clock_policy {
13        // Desc: number of cpu cycles since reset (or return from hibernation).
14        // Note: be aware of std::chrono::system_clock in TR1.
15        // Ref:  1) "Using the RDTSC Instruction for Performance Monitoring"
16        //          Intel (1997), www.ccsl.carleton.ca/~jamuir/rdtscpm1.pdf
17        //       2) www.intel.com/products/processor/manuals/
18
19        typedef /*? unsigned */ long long counter_type;
20
21  #ifdef _MSC_VER
22  #ifdef _M_X64
23        // Ref: msdn.microsoft.com/en-us/library/twchhe95.aspx on intrinsic __rdtsc.
24        //typedef __int64 counter_type;
25        static counter_type current() { return __rdtsc(); }
26        static std::string unit_name() { return "clock cycles"; }
```

```
27  #else
28  #ifdef _M_IX86
29  #if (_M_IX86 >= 500) // instruction set >= Intel Pentium I.
30        static counter_type current() {
31          assert(sizeof(__int64) == sizeof(long long));
32          long long cpu_c;
33          __asm {
34            // optional begin
35            //push ebx
36            //push ecx
37            //cpuid // id -> (ebx,ebc). side effect: serialize execution.
38            //pop ecx
39            //pop ebx
40            // optional end
41            push eax
42              push edx
43              rdtsc // read time-stamp counter
44              mov dword ptr [cpu_c],eax
45              mov dword ptr [cpu_c+4],edx
46              pop edx
47              pop eax
48          }
49          return cpu_c;
50        }
51        static std::string unit_name() { return "clock cycles"; }
52  #else // old Intel
53        static counter_type current() { return faster_clock_policy::current(); }
54        static std::string unit_name() { return faster_clock_policy::unit_name(); }
55  #endif
56  #else // non Intel
57        static counter_type current() { return faster_clock_policy::current(); }
58        static std::string unit_name() { return faster_clock_policy::unit_name(); }
59  #endif
60  #endif
61  #elif defined(__GNUC__)
62  #if defined(__i386__)
63        static __inline__ counter_type current() {
64          counter_type cpu_c;
65          __asm__ volatile (".byte 0x0f, 0x31" : "=A" (cpu_c));
66          return cpu_c;
67        }
68        static std::string unit_name() { return "clock cycles"; }
69  #elif defined(__x86_64__)
70        static __inline__ volatile long long current() {
71          register long long cpu_c asm("eax"); // edx:eax
72          asm volatile (".byte 15, 49" : : : "eax", "edx");
73          return cpu_c;
74        }
75        static std::string unit_name() { return "clock cycles"; }
76  #else // cpu
77        static counter_type current() { return faster_clock_policy::current(); }
78        static std::string unit_name() { return faster_clock_policy::unit_name(); }
79  #endif
80  #else // other compiler
81        static counter_type current() { return faster_clock_policy::current(); }
82        static std::string unit_name() { return faster_clock_policy::unit_name(); }
83  #endif
84
85        typedef counter_type hertz;
86
87        static hertz frequency() {
88          if(frequency_ == 0)
89            frequency_ = std::max(
90            probe_frequency()
91            ,probe_frequency()
92            );
93          assert(frequency_ > 0);
94          return frequency_;
95        }
96        static hertz probe_frequency() {
97          faster_clock_policy::counter_type
98            wait(faster_clock_policy::frequency() / 4);
99          faster_clock_policy::counter_type
100           f(faster_clock_policy::current()+wait);
```

```
101
102           counter_type t1(current());
103           while(faster_clock_policy::current() < f) ;
104           counter_type t2(current());
105
106           return (t2-t1)*4;
107         }
108       private:
109         static hertz frequency_;
110       };
111       tsc_clock_policy::hertz tsc_clock_policy::frequency_(0);
112     }
113   }
114   #endif
```

C.13 gnuplot_filter.cpp

```
1   /*
2   * Desc: Worst case data filtering for gnuplot-scripts.
3   * Auth: Asger Bruun 2009
4   */
5
6   #include <regex>
7   #include <iostream>
8   #include <sstream>
9   #include <string>
10  #include <cstddef> // std::size_t
11
12  #include "ms_c_fix.hpp"
13
14  using namespace std;
15  using namespace cphstl;
16
17  template<int N>
18  bool pass(std::size_t s) {
19    return true;
20  }
21
22  template<>
23  bool pass<1>(size_t s) {
24    return cphstl::pop_cnt(s+1) == 1;
25  }
26
27  template<>
28  bool pass<2>(size_t s) {
29    size_t succ(s+1), l, m;
30    switch(cphstl::pop_cnt(succ)) {
31      case 1: return true; break;
32      case 2:
33        l = trailing_zeros(succ);
34        m = bit_scan_reverse(succ);
35        return (m-l) == 1;
36        break;
37      default: return false;
38    }
39  }
40
41  template<int N>
42  int process() {
43    const std::tr1::regex rx("(\\w*)(\\d+)(\\w+)(.*)");
44    string s;
45    while(getline(cin, s)) {
46      if(regex_match(s.begin(), s.end(), rx)) {
47        std::stringstream ss(s);
48        long i;
49        ss >> i;
50        if(pass<N>(i)) cout << s << endl;
51        // else cout << "# skip: " << i << endl;
52      }
53      else cout << s << endl;
54    }
55    return 0;
56  }
```

```
57
58
59  int main(int argc, char** argv) {
60    int filter(0);
61    if(argc>1) {
62      std::stringstream ss(argv[1]);
63      ss >> filter;
64    }
65    switch(filter) {
66      case 1: return process<1>(); break;
67      case 2: return process<2>(); break;
68      default:
69        std::cerr << "\n\n#-----␣gnuplot␣data␣filtering␣utility␣-----\n";
70        std::cerr << "arguments:␣[gnuplot_data_filename]␣[data_selection]\n"
71          << "\t␣data_selection␣1:␣2^k-1.\n"
72          << "sample:␣'gnu.dat␣2'.\n";
73        return 1;
74    }
75  }
```

D Benchmark

D.1 benchmark.i++

```
#ifndef _CPHSTL_BENCHMARK_I_
#define _CPHSTL_BENCHMARK_I_

/*
Desc: The Benchmarks.
:     Imports the original PQFW benchmarks, but rewritten.
Auth: Asger Bruun 2009-2010.
*/

#include "ms_c_fix.hpp"
#include "bit_manipulation.h++"
#include "benchmark_main.hpp"

namespace cphstl {
  namespace benchmarking {

    template <typename Q, typename Counter /* = combi_counter*/>
    struct pq_benchmark : public benchmark_suite<Counter> {
      typedef typename benchmark_suite<Counter>::size_type size_type;
      typedef typename benchmark_suite<Counter>::benchmark benchmark;
      typedef typename Q::value_type value_type;
      typedef typename Q::iterator iterator;

      enum { number_system = Q::realizator_type::number_system };
      /*
      1: ir - indirect redundant binary
      2: db - direct binary
      3: lc - direct redundant binary
      */


      struct benchmark_data {
        std::vector<value_type> a;

        benchmark_data(size_type const& element_count
          , size_type const& repetition) : a(element_count)
        {
          for (unsigned int i = 0; i != element_count; ++i)
            a[i] = value_type(i);

          if(randomise_test_data) // randomise every repetition,
            srand((unsigned int) repetition);
          else srand(1); // or use this for same random data.

          for (unsigned int i = 0; i != element_count; ++i)
            std::swap(a[i], a[rand() % (element_count)]);
        }

        ~benchmark_data() {
          a.clear();
        }
      };


      struct push_n : public benchmark {
        std::string name() { return "push_n"; }

        void run(size_type const& element_count, size_type const& repetition) {
          benchmark_data d(element_count, repetition);
          Q q;
          (*this).counter.start(element_count);
          for (unsigned int i = 0; i != element_count; ++i) {
            (void) q.push(d.a[i]);
          }
          (*this).counter.stop();
        }
      };

      struct increase_n : public benchmark {
```

```
70          std::string name() { return "increase_n"; }
71
72          void run(size_type const& element_count, size_type const& repetition) {
73            benchmark_data d(element_count, repetition);
74            Q q;
75            std::vector<iterator> v;
76            for (unsigned int i = 0; i != element_count; ++i) {
77              iterator p = q.push(d.a[i]);
78              v.push_back(p);
79            }
80            // !!! 20100324 leda cant: value_type m(*q.top());
81            value_type m(element_count);
82            (*this).counter.start(element_count);
83            for (unsigned int i = 0; i != element_count; ++i) {
84              q.increase(v[i], ++m);
85              //dont: top() could have side effects: q.increase(v[i], *v[i] + *q.top() + 1);
86              assert(q.top() == v[i]);
87            }
88            (*this).counter.stop();
89          }
90        };
91
92        struct erase_n : public benchmark {
93          std::string name() { return "erase_n"; }
94          std::string description() { return "erase_n - extract(p)"; }
95
96          void run(size_type const& element_count, size_type const& repetition) {
97            benchmark_data d(element_count, repetition);
98            Q q;
99            std::vector<iterator> v;
100           for (unsigned int i = 0; i != element_count; ++i) {
101             iterator p = q.push(d.a[i]);
102             v.push_back(p);
103           }
104
105           (*this).counter.start(element_count);
106           for (unsigned int i = 0; i != element_count; ++i) {
107             q.erase(v[i]);
108           }
109           (*this).counter.stop();
110           assert(q.empty());
111         }
112
113         void gnuplot_data() {
114           if(number_system == 2)
115           (*this).counter.template gnuplot_data<typename special_sizes
116           ::template post_processor<typename special_sizes::size_x1_1> >(name());
117           else
118             benchmark::gnuplot_data();
119         }
120       };
121       //struct erase_one : public benchmark {
122       //  // Desc: erase one random element from n elements.
123       //  std::string name() { return "erase_one"; }
124
125       //  erase_one() : benchmark(true /*per test*/) {}
126
127       //  void run(size_type const& element_count, size_type const& repetition) {
128       //    iterator t;
129       //    Q p;
130       //    {
131       //      benchmark_data c(element_count, repetition);
132       //      for (unsigned int i = 0; i != element_count; ++i) {
133       //        if(i == 0) t = p.push(c.a[i]);
134       //        else (void) p.push(c.a[i]);
135       //      }
136       //    }
137       //    (*this).counter.start(element_count);
138       //    (void) p.erase(i);
139       //    (*this).counter.stop();
140       //  }
141
142       //  void gnuplot_data() {
143       //    if(number_system == 2)
```

```
144         //     (*this).counter.template gnuplot_data<typename special_sizes
145         //     ::template post_processor<typename special_sizes::size_x1_1> >(name());
146         //     else
147         //       benchmark::gnuplot_data();
148         //   }
149         //};
150
151         struct pop_n : public benchmark {
152           std::string name() { return "pop_n"; }
153           std::string description() { return "pop_n - extract(find_top())"; }
154
155           void run(size_type const& element_count, size_type const& repetition) {
156             benchmark_data d(element_count, repetition);
157             Q q;
158             for (unsigned int i = 0; i != element_count; ++i) {
159               (void) q.push(d.a[i]);
160             }
161             (*this).counter.start(element_count);
162             for (unsigned int i = 0; i != element_count; ++i) {
163               q.pop();
164             }
165             (*this).counter.stop();
166           }
167         };
168         struct pop_one : public benchmark {
169           // Desc: try this on binary numbers for elements: 2^k - 1.
170           std::string name() { return "pop_one"; }
171
172           pop_one() : benchmark(true /*per test*/) {}
173
174           void run(size_type const& element_count, size_type const& repetition) {
175             Q p;
176             {
177               benchmark_data c(element_count, repetition);
178               for (unsigned int i = 0; i != element_count; ++i) {
179                 (void) p.push(c.a[i]);
180               }
181             }
182             (*this).counter.start(element_count);
183             (void) p.pop();
184             (*this).counter.stop();
185           }
186
187           void gnuplot_data() {
188             if(number_system == 2)
189             (*this).counter.template gnuplot_data<typename special_sizes
190             ::template post_processor<typename special_sizes::size_x1_1> >(name());
191             else
192               benchmark::gnuplot_data();
193           }
194         };
195 /* this test was naive:
196         struct pop_push_n : public benchmark {
197           // Desc: first attempt to measure Vuillemin push on size with all bits set.
198           std::string name() { return "pop_push_n"; }
199
200           void run(size_type const& element_count, size_type const& repetition) {
201             Q p;
202             {
203               benchmark_data c(element_count, repetition);
204               for (unsigned int i = 0; i != element_count; ++i) {
205                 (void) p.push(c.a[i]);
206               }
207             }
208             benchmark_data d(element_count, repetition+1);
209
210             (*this).counter.start(element_count);
211             for (unsigned int i = 0; i != element_count; ++i) {
212               (void) p.pop();  // size all ones if was pow of 2.
213               (void) p.push(d.a[i]);
214             }
215             (*this).counter.stop();
216           }
217         };
```

```
218 */
219       struct push_one : public benchmark {
220         // Desc: second attempt to measure Vuillemin push on size with all bits set,
221         //       eliminating the noise from pop, but in much lower precision.
222         std::string name() { return "push_one"; }
223
224         push_one() : benchmark(true /*per test*/) {}
225
226         void run(size_type const& element_count, size_type const& repetition) {
227           Q p;
228           {
229             benchmark_data c(element_count, repetition);
230             for (unsigned int i = 0; i != element_count; ++i) {
231               (void) p.push(c.a[i]);
232             }
233           }
234           // !!! 20100324 leda cant: value_type m(1 + *p.top());
235           value_type m(1 + element_count);
236           // size all ones if pow of 2 - 1.
237           //assertion_help(if(element_count == 282) __break_here;)
238
239           (*this).counter.start(element_count);
240           (void) p.push(m);
241           (*this).counter.stop();
242         }
243
244         void gnuplot_data() {
245           if(number_system == 2)
246           (*this).counter.template gnuplot_data<typename special_sizes
247           ::template post_processor<typename special_sizes::size_x1_1> >(name());
248           else
249             benchmark::gnuplot_data();
250         }
251       };
252
253       struct push_last_one : public benchmark {
254         // Desc: second attempt to measure Vuillemin push on size with all bits set,
255         //       eliminating the noise from pop, but in much lower precision.
256         std::string name() { return "push_last_one"; }
257
258         push_last_one() : benchmark(true /*per test*/) {}
259
260         void run(size_type const& element_count, size_type const& repetition) {
261           Q p;
262           value_type m;
263           {
264             benchmark_data c(element_count, repetition);
265             for (unsigned int i = 0; i != (element_count-1); ++i) {
266               (void) p.push(c.a[i]);
267             }
268             m = c.a[(element_count-1)];
269           }
270           // !!! 20100324 leda cant: value_type m(1 + *p.top());
271           // size all ones if pow of 2 - 1.
272           //assertion_help(if(element_count == 282) __break_here;)
273
274           (*this).counter.start(element_count);
275           (void) p.push(m);
276           (*this).counter.stop();
277         }
278
279         void gnuplot_data() {
280           if(number_system == 2)
281           (*this).counter.template gnuplot_data<typename special_sizes
282           ::template post_processor<typename special_sizes::size_x1_1> >(name());
283           else
284             benchmark::gnuplot_data();
285         }
286       };
287
288       struct erase_one : public benchmark {
289         std::string name() { return "erase_one_leaf"; }
290         std::string description() { return "erase_one_leaf - extract(p)"; }
291
```

```
292           erase_one() : benchmark(true /*per test*/) {}
293
294           void run(size_type const& element_count, size_type const& repetition) {
295             benchmark_data d(element_count, repetition);
296             iterator m;
297             Q q;
298             std::vector<iterator> v;
299             for (unsigned int i = 0; i != element_count; ++i) {
300               iterator p = q.push(d.a[i]);
301               v.push_back(p);
302               if((i == 0) || (d.a[i] == 0)) m = p;
303             }
304
305             (*this).counter.start(element_count);
306             //(void) q.erase(v[repetition]); // error was p!!!
307             (void) q.erase(m);
308             (*this).counter.stop();
309           }
310
311           void gnuplot_data() {
312             if(number_system == 2)
313             (*this).counter.template gnuplot_data<typename special_sizes
314             ::template post_processor<typename special_sizes::size_x1_1> >(name());
315             else
316               benchmark::gnuplot_data();
317           }
318         };
319
320
321         struct increase_lowest_one_to_top : public benchmark {
322           std::string name() { return "increase_lowest_one_to_top"; }
323
324           increase_lowest_one_to_top () : benchmark(true /*per test*/) {}
325
326           void run(size_type const& element_count, size_type const& /*repetition*/) {
327             Q p;
328             iterator e(p.push(0)); // 1 +
329             for (unsigned int i(1); i != element_count; ++i) { // n - 1 = 1.
330               (void) p.push(i);
331             }
332             //iterator top(p.top());
333             // !!! 20100324 leda cant: assert((*top) > (*e)); // suitable ordering.
334             // size all ones if pow of 2 - 1.
335             (*this).counter.start(element_count);
336             (void) p.increase(e, element_count);
337             (*this).counter.stop();
338           }
339
340           void gnuplot_data() {
341             if(number_system == 2)
342             (*this).counter.template gnuplot_data<typename special_sizes
343             ::template post_processor<typename special_sizes::size_x1_1> >(name());
344             else
345               benchmark::gnuplot_data();
346           }
347         };
348
349         struct meld : public benchmark {
350           std::string name() { return "meld"; }
351
352           meld () : benchmark(true /* single op*/) {}
353
354           void run(size_type const& element_count, size_type const& repetition) {
355             benchmark_data d(element_count, repetition+1);
356             Q p;
357             {
358               //benchmark_data c(element_count / 2, repetition); // 20100725 changed to fit
                      max mem better.
359               // benchmark_data c(element_count * 2, repetition);
360               // ??? better:            benchmark_data c(element_count * element_count,
                      repetition);
361
362               for (unsigned int i = 0; i != element_count; ++i) {
363               //for (unsigned int i = 0; i != element_count / 2; ++i) {
```

```
364             //for (unsigned int i = 0; i != element_count *2; ++i) {
365               (void) p.push(2*d.a[i] +1); // odd
366             }
367           }
368           Q q;
369           for (unsigned int i = 0; i != element_count; ++i) {
370             (void) q.push(2 * d.a[element_count-(i+1)]); // even
371           }
372           (*this).counter.start(element_count);
373           p.meld(q);
374           (*this).counter.stop();
375         }
376       };
377
378       struct clear : public benchmark {
379         std::string name() { return "clear"; }
380         std::string description() { return "clear"; }
381
382         void run(size_type const& element_count, size_type const& repetition) {
383           benchmark_data d(element_count, repetition);
384           Q q;
385           for (unsigned int i = 0; i != element_count; ++i) {
386             (void) q.push(d.a[i]);
387           }
388           (*this).counter.start(element_count);
389           q.clear();
390           (*this).counter.stop();
391         }
392
393         void gnuplot_data() {
394           if(number_system == 1)
395           (*this).counter.template gnuplot_data<typename special_sizes
396           ::template post_processor<typename special_sizes::size_pow_n_plus_pred_n> >(name()
                  );
397           else
398             benchmark::gnuplot_data();
399         }
400       };
401
402       //struct erase_any_n : public benchmark {
403       //  std::string name() { return "erase_any_n"; }
404       //  std::string description() { return "erase_any_n - extract()"; }
405
406       //  void run(size_type const& element_count, size_type const& repetition) {
407       //    benchmark_data d(element_count, repetition);
408       //    Q q;
409       //    for (unsigned int i = 0; i != element_count; ++i) {
410       //      (void) q.push(d.a[i]);
411       //    }
412       //    (*this).counter.start(element_count);
413       //    for (unsigned int i = 0; i != element_count; ++i) {
414       //      q.erase(v[i]);
415       //    }
416       //    q.clear();
417       //    (*this).counter.stop();
418       //  }
419
420       //  void gnuplot_data() {
421       //    if(number_system == 1)
422       //    (*this).counter.template gnuplot_data<typename special_sizes
423       //    ::template post_processor<typename special_sizes::size_pow_n_plus_pred_n> >(
            name());
424       //    else
425       //      benchmark::gnuplot_data();
426       //  }
427       //};
428
429       struct copy : public benchmark {
430         std::string name() { return "copy"; }
431         std::string description() { return "copy - constructor"; }
432
433         void run(size_type const& element_count, size_type const& repetition) {
434           benchmark_data d(element_count, repetition);
435           Q q;
```

```
436             for (unsigned int i = 0; i != element_count; ++i) {
437               (void) q.push(d.a[i]);
438             }
439             (*this).counter.start(element_count);
440             {
441               Q r(q);
442               (*this).counter.stop();
443             }
444           }
445
446           void gnuplot_data() {
447             if(number_system == 1)
448             (*this).counter.template gnuplot_data<typename special_sizes
449             ::template post_processor<typename special_sizes::size_pow_n_plus_pred_n> >(name()
                  );
450             else
451               benchmark::gnuplot_data();
452           }
453         };
454
455
456         struct stefan : public benchmark {
457           std::string name() { return "stefan"; }
458           std::string description() { return "Stefan - a mixed benchmark"; }
459           // Pseudo:
460           //   N x insert()
461           //   N x increase()
462           //   N/2 x extract(p)
463           //   N/2 x pop()
464           void run(size_type const& element_count, size_type const& repetition) {
465             benchmark_data d(element_count, repetition);
466             Q q;
467             (*this).counter.start(element_count);
468
469             std::vector<iterator> v;
470             for (unsigned int i = 0; i != element_count; ++i) {
471               iterator p = q.push(d.a[i]);
472               v.push_back(p);
473             }
474             // !!! 20100324 leda can't: value_type m(*q.top());
475             value_type m(element_count);
476             for (unsigned int i = 0; i != element_count/2; ++i) {
477               // !!! 20100324 leda cant: q.increase(v[i], *v[i] + (++m));
478               q.increase(v[i], ++m);
479             }
480             for (unsigned int i = 0; i != element_count/2; ++i) {
481               q.erase(v[i]);
482             }
483             for (unsigned int i = 0; i != element_count/2; ++i) {
484               q.pop();
485             }
486             (*this).counter.stop();
487           }
488         };
489
490         pq_benchmark(std::string impl_name, string_filter & methods_filter)
491           : benchmark_suite<Counter>(impl_name, methods_filter)
492         {
493           (*this).template registrate<push_n>("push_n");
494           (*this).template registrate<increase_n>("increase_n");
495           (*this).template registrate<erase_n>("erase_n");
496           (*this).template registrate<erase_one>("erase_one_leaf");
497           (*this).template registrate<pop_n>("pop_n");
498           (*this).template registrate<pop_one>("pop_one");
499           //(*this).template registrate<pop_push_n>("pop_push_n");
500           (*this).template registrate<push_one>("push_one");
501           (*this).template registrate<push_last_one>("push_last_one");
502           (*this).template registrate<increase_lowest_one_to_top
503           >("increase_lowest_one_to_top");
504           (*this).template registrate<meld>("meld");
505           //(*this).template registrate<erase_any_n>("erase_any_n");
506           (*this).template registrate<clear>("clear");
507           (*this).template registrate<copy>("copy");
508           (*this).template registrate<stefan>("stefan");
```

```
509         }
510
511         void type_report() {
512           std::cout << "\n" << (*this).friendly_name << "␣queue␣type:␣";
513           std::cout << typeid_name_fix(typeid(Q)) << "\n";
514           std::cout << "\n" << (*this).friendly_name << "␣realizator␣type:␣";
515           std::cout << typeid_name_fix(typeid(typename Q::realizator_type)) << "\n\n";
516         }
517
518         struct special_sizes {
519           struct size_x1_1 {
520             static bool focus(size_type s) {
521               // 2^21 - 1, 2^21 + 2^20 - 1, 2^22 - 1,..
522               size_type succ(s+1), l, m;
523               switch(cphstl::pop_cnt(succ)) {
524               case 1: return true; break;
525               case 2:
526                 l = cphstl::trailing_zeros(succ);
527                 m = cphstl::bit_scan_reverse(succ);
528                 return (m-l) == 1;
529                 break;
530               default: return false;
531               }
532             }
533           };
534
535           struct size_pow_n_plus_pred_n {
536             static bool focus(size_type s) {
537               // 2^20 + 19, 2^21 + 20, 2^22 + 21,.. (for weak queue clear)
538               size_type m(bit_scan_reverse(s));
539               if(m<2) return false;
540               return s == (size_type(1) << m) + m - 1;
541             }
542           };
543
544           /*
545           bool focus<size_1_1>(size_type s) {
546             // 2^21 - 1, 2^22 - 1,..
547             return cphstl::pop_cnt(s+1) == 1;
548           }
549           bool focus<size_pow_n_plus_pow_pred_n>(size_type s) {
550             // 2^21 + 2^20, 2^22 + 2^21,..
551             size_type succ(s+1), l, m;
552             switch(cphstl::pop_cnt(succ)) {
553               case 1: return false; break;
554               case 2:
555                 l = trailing_zeros(succ);
556                 m = bit_scan_reverse(succ);
557                 return (m-l) == 1;
558                 break;
559               default: return false;
560             }
561           }
562           */
563
564           template <typename S>
565           struct post_processor {
566             static void process(size_type const& count, data_columns::row_type & row) {
567               assert(data_columns::column_count == row.size());
568               row[data_columns::user_def] = S::focus(count)?2:1;
569             }
570           };
571         };
572
573         bool focus(size_type const& elements) {
574           return special_sizes::size_x1_1::focus(elements)
575             || special_sizes::size_pow_n_plus_pred_n::focus(elements);
576         }
577       };
578     } // namespace benchmarking
579   } // namespace cphstl
580   #endif
```

D.2 makefile.mk

```
# Desc: Benchmark on selected priority queue combinations.
#       High granularity tests capable at searching for ressource consumption peaks.
#       Outputs the raw data and plots: simple, box and whiskers and best values.
# Auth: Asger Bruun 2010.


# CPHSTL_ROOT=$(HOME)/CPHSTL
# CPHSTL_ROOT=/mnt/hgfs/cphstl
CPHSTL_ROOT=/media/win/cvs/CPHSTL

ASGER_ROOT=$(CPHSTL_ROOT)/Progress/Meldable-priority-queue/Asger-weak-binomial-mixed
CPHSTL_SOURCE=$(ASGER_ROOT)/CPHSTL_Branch

CXXFLAGS = -DNDEBUG -Wall -std=c++0x -pedantic -x c++ -fno-strict-aliasing -O3
TEST_CXXFLAGS = -Wall -std=c++0x -pedantic -x c++ -fno-strict-aliasing
PROFILING_CXXFLAGS = -DNDEBUG -DPROFILE -Wall -std=c++0x -pedantic -x c++ -fno-strict-aliasing
      -O3 -g

IFLAGS = -I . -I ../Code -I ../Test -I ../Benchmarking -I ../Code/CPHSTL_modified -I "$(
     CPHSTL_SOURCE)/Iterator/Code" -I "$(CPHSTL_SOURCE)/Common/Code" -I "$(CPHSTL_SOURCE)/Type/
     Code" -I "$(CPHSTL_SOURCE)/Priority-queue-frameworks/Code" -I "$(CPHSTL_SOURCE)/Proxy/Code
     " -I "$(CPHSTL_SOURCE)/Meldable-priority-queue/Code" -I "$(LEDAROOT)/incl" -L$(LEDAROOT)

LFLAGS = -lleda -lm -lX11
CXX = g++


# implementations:
IMPLS=binary_number_PWH,redundant_number_PWH,redundant_number_R,redundant_number_K,
     new_weak_queue,weak_queue

SELECTED_METHODS=push_n increase_n erase_n erase_one_leaf pop_n pop_one push_one push_last_one
     increase_lowest_one_to_top meld clear stefan

# Command line for profiler (if invoked):
PROF_CMD="stefan" "binary_number_PWH,redundant_number_R,redundant_number_K" "pro" 6 7 3

# execution schedule (max power of two, number of intervals between every power of two,
     minimum number of repeats for large data counts):
REP=23 7 9

OUT_DIR=./linux_gcc


%.png: %.eps
  gs -dNOPAUSE -dBATCH -sDEVICE=png256 -r600 -dEPSCrop -sOutputFile="$@" "$<"

%.pdf: %.eps
  gs -dNOPAUSE -dBATCH -sDEVICE=pdfwrite dEPSCrop -sOutputFile="$@" "$<"

.PHONY : $(SELECTED_METHODS) all dir_out executable methods png test leak-check prof gprof
     clean

all: methods png

dir_out:
  @if [ ! -d "$(OUT_DIR)" ]; then mkdir "$(OUT_DIR)"; fi

executable:
  $(CXX) $(CXXFLAGS) $(IFLAGS) pq_benchmark.cpp $(LFLAGS)

methods: $(SELECTED_METHODS)

$(SELECTED_METHODS): dir_out executable
  ./a.out "$@" "$(IMPLS)" gnu $(REP) > "$(OUT_DIR)/$@.gnu"
  # ./a.out "$@" "$(IMPLS)" rep $(REP) > "$(OUT_DIR)/$@_report.txt"
  ./a.out "$@" "$(IMPLS)" typ $(REP) > "$(OUT_DIR)/$@_types.txt"
  ./a.out "$@" "$(IMPLS)" dat $(REP) > "$(OUT_DIR)/$@.dat"
  @echo "1 0 0 0 0 0 2" > $(OUT_DIR)/dummy.dat
  @echo "end" >> $(OUT_DIR)/dummy.dat
  cd "$(OUT_DIR)"; \
  gnuplot "$@.gnu"; \
```

```
65    cd ./..
66
67  leak-check: executable
68    # Note: eager-mark-store.h++ plus LEDA are leaking.
69    valgrind --leak-check=full --show-reachable=yes ./a.out . . tst 6
70
71  test: dir_out
72    $(CXX) $(TEST_CXXFLAGS) $(IFLAGS) pq_benchmark.cpp $(LFLAGS) -o test.out
73    ./test.out . . tst 6 > $(OUT_DIR)/test.txt
74    ./test.out . . typ 6 > $(OUT_DIR)/test_types.txt
75    ./test.out "copy" "binary_number_PWH,redundant_number_R,redundant_number_K,new_weak_queue,
          weak_queue" rep 18 17 7
76    # adjust exponent 6 according to your machine and patience (usually i use 10).
77
78  prof: dir_out
79    $(CXX) $(CXXFLAGS) -DPROFILE -g $(IFLAGS) pq_benchmark.cpp $(LFLAGS) -o prof.out
80    valgrind --tool=callgrind --simulate-cache=yes --dump-instr=yes --collect-jumps=yes --instr-
          atstart=no ./prof.out $(PROF_CMD) > /dev/null
81
82  gprof: dir_out
83    $(CXX) $(CXXFLAGS) -pg -g $(IFLAGS) pq_benchmark.cpp $(LFLAGS) -o gprof.out
84    ./gprof.out $(PROF_CMD) > /dev/null
85    gprof gprof.out > $(OUT_DIR)/gprofile.txt
86
87  png:
88    cd $(OUT_DIR); \
89    for f in *.eps; \
90    do gs -dNOPAUSE -dBATCH -sDEVICE=png256 -r600 -dEPSCrop -sOutputFile="$$f.png" "$$f"; \
91    done; \
92    cd ./..
93
94  pdf:
95    cd $(OUT_DIR); \
96    for f in *.eps; \
97    do gs -dNOPAUSE -dBATCH -sDEVICE=pdfwrite -dEPSCrop -sOutputFile="$$f.png" "$$f"; \
98    done; \
99    cd ./..
100
101 clean:
102   @rm -vf *~ a.out test.out gprof.out prof.out core
103   @rm -vf $(OUT_DIR)/*
104   @rmdir $(OUT_DIR)
```

D.3 make_benchmark.cmd

```
1   REM *** Run the benchmarks and produce the plots
2
3   REM # output directory:
4   set out_dir=w7_i7_msvc
5
6   REM # methods:
7   set meths=push_n,increase_n,erase_n,erase_one_leaf,pop_n,pop_one,push_one,push_last_one,
        increase_lowest_one_to_top,meld,clear,stefan
8
9   REM # implementations (most of them are disabled with REM):
10  set impls=
11  REM set impls=%impls%,db_bno_3_dv,db_pwh_3_dv,db_bno_2_dv,db_bno_r_dv,db_bno_k_dv,
        db_bno_3_iv,db_pwh_3_iv,db_bno_3_dv_slow_top,db_pwh_3_dv_slow_top,dl(u)_bno_3_dv,dl(m
        )_bno_3_dv,dl(f)_bno_3_dv,dl(u)_pwh_3_dv,dl(m)_pwh_3_dv,dl(f)_pwh_3_dv,dl(u)_bno_r_dv
        ,dl(u)_bno_k_dv,dl(f)_bno_r_dv,dl(f)_bno_k_dv,ir_pwh_3_dv,ir_bno_3_dv,ir_pwh_3_iv,
        ir_bno_3_iv,ir_bno_3_proxy
12  REM set impls=%impls%,leda_fibonacci_heap,leda_pairing_heap
13  REM set impls=%impls%,db_pwh_3_dv_slow_top
14  REM set impls=%impls%,pennant_queue,weak_heap
15  set impls=%impls%,new_weak_queue,weak_queue
16  set impls=%impls%,binary_number_PWH,redundant_number_PWH,redundant_number_R,redundant_number_K
17
18  REM # execution schedule (max power of two, number of intervals between every power of
        two, minimum number of repeats for large data counts):
19
20  REM set count=25 3 7
21  REM set count=20 117 9
22  REM set count=20 17 9
```

```
23  set count=24 7 9
24
25  REM # directory of executable:
26  set ver=Release
27  REM # only for test: set ver=Debug
28
29
30  if "%1"=="clean" goto clean
31  if "%1"=="gnuplot" goto gnuplot
32  if "%1"=="all" goto benchmark
33  if "%1"=="" goto benchmark
34
35  rem Use:
36  rem    make all       - (default) run benchmark and gnuplot
37  rem    make clean     - remove gnuplot output
38  rem    make benchmark - run benchmarks
39  rem    make gnuplot   - gnuplot output
40  pause
41  goto end
42
43  :clean
44  cd /d "%~dp0%out_dir%"
45  for %%e in (png,pdf,eps,ps) do del *.%%e
46  goto end
47
48  :benchmark
49  if not exist %~dp0%out_dir%\*.* md %~dp0%out_dir%
50  cd /d "%~dp0"
51  for %%m in (%meths%) do ..\Test\vs2008\%ver%\pq_benchmark.exe %%m %impls% gnu %count% > %
        out_dir%\%%m.gnu
52  for %%m in (%meths%) do ..\Test\vs2008\%ver%\pq_benchmark.exe %%m %impls% typ %count% > %
        out_dir%\%%m.typ
53  for %%m in (%meths%) do ..\Test\vs2008\%ver%\pq_benchmark.exe %%m %impls% dat %count% > %
        out_dir%\%%m.dat
54  if errorlevel 1 goto bench_error
55  call :gnuplot %1 %2 %3 %4 %5 %6 %8 %9
56  goto end
57  :bench_error
58  pause
59  goto end
60
61  :gnuplot
62  cd /d "%~dp0%out_dir%"
63  if errorlevel 1 pause
64  echo   1 0 0 0 0 0 0 4 > dummy.dat
65  echo end >> dummy.dat
66  for %%m in (%meths%) do "%userprofile%\gnuplot\binaries\gnuplot.exe" %%m.gnu
67  if errorlevel 1 pause
68  set path=%path%;C:\Program Files\gs\gs8.64\bin
69  set path=%path%;C:\Program Files\gs\gs8.64\lib
70  REM # png
71  for %%f in (*.ps) do gswin32c.exe -dNOPAUSE -dBATCH -sDEVICE=png256 -r600 -sOutputFile="%%~nf.
        png" "%%f"
72  for %%f in (*.eps) do gswin32c.exe -dNOPAUSE -dBATCH -sDEVICE=png256 -r600 -dEPSCrop -
        sOutputFile="%%~nf.png" "%%f"
73  if errorlevel 1 pause
74  REM # pdf
75  for %%f in (*.ps) do gswin32c.exe -dNOPAUSE -dBATCH -sDEVICE=pdfwrite -sOutputFile="%%~nf.pdf"
        "%%f"
76  for %%f in (*.eps) do gswin32c.exe -sDEVICE=pdfwrite -dEPSCrop -o "%%~nf.pdf" "%%f"
77  REM # above is better than: for %%f in (*.eps) do ps2pdf.exe  "%%f" "%%~nf.pdf"
78  if errorlevel 1 pause
79  goto end
80
81  :end
```

D.4 pq_benchmark.cpp

```
1  #define INCLUDE_LEAK_CHECK
2
3  #include "ms_c_fix.hpp"
4  #include "assert.h++"
5
```

```
#include <iostream>

const bool careful(true); // needs to be true for cphstl_extract

#ifndef NDEBUG // manual PQFW switch
#define DEBUG
#endif

#include "heap_store_config.hpp"
#include "heap_store_config_alt.hpp"

#include "benchmarking.hpp"
#include "benchmark.i++"

struct impl_translation {
  static std::string translate(std::string const& desc) {
    std::string name(str_replace(desc,"_","-")), s(name);
    const bool danish(false);
    if(danish) {
      s = str_replace(s, "ir", "indirekte binær redundant skov med");
      s = str_replace(s, "db", "direkte binær skov med");
      s = str_replace(s, "dl(", "direkte doven mente(");
      s = str_replace(s, "(u)", " skov med letvægt join schedule og");
      s = str_replace(s, "(f)", " skov med tung join schedule og");
      s = str_replace(s, "pwh", "svage træer");
      s = str_replace(s, "-bno-r-", "-binomiale Brown R træer-");
      s = str_replace(s, "-bno-k-", "-binomiale Brown K træer-");
      s = str_replace(s, "bno", "binomiale træer");
      s = str_replace(s, "-3", "");
      s = str_replace(s, "2", "2 peger");
      s = str_replace(s, "-dv", "");
      s = str_replace(s, "-iv", "-med indirekte værdi");
      s = str_replace(s, "slow-top", "og langsom top");
      s = str_replace(s, "-", " ");
      if(name == "ir-pwh-3-dv") s += " = std svag kø";
      else if(name == "db-bno-3-dv") s += " = std binomial kø";
      else if(name == "db-pwh-3-dv") s += " = Vuillemins forslag til implementering";
      s = str_replace(s, "æ", "ae");
      s = str_replace(s, "ø", "oe");
      s = str_replace(s, "å", "aa");
    } else {
      s = str_replace(s, "ir", "indirect redundant binary forest with");
      s = str_replace(s, "db", "direct binary forest with");
      s = str_replace(s, "dl(", "direct redundant binary(");
      s = str_replace(s, "(u)", " forest with ultralight join schedule and");
      s = str_replace(s, "(f)", " forest with fat join schedule and");
      s = str_replace(s, "pwh", "weak trees");
      s = str_replace(s, "-bno-r-", "-binomial Brown R trees-");
      s = str_replace(s, "-bno-k-", "-binomial Brown K trees-");
      s = str_replace(s, "bno", "binomial trees");
      s = str_replace(s, "-3", "");
      s = str_replace(s, "2", "2 pointer");
      s = str_replace(s, "-dv", "");
      s = str_replace(s, "-iv", "-with indirect value");
      s = str_replace(s, "slow-top", "and slow top");
      s = str_replace(s, "-", " ");
      if(name == "ir-pwh-3-dv") s += " = std weak queue";
      else if(name == "db-bno-3-dv") s += " = std binomial queue";
      else if(name == "db-pwh-3-dv") s += " = Vuillemin implementation";
    }
    //return name + " - " + s;
    return name;
  }
};

int run(int argc, char** argv) {
  std::cout << "# command: \"";
  for(int i = 0; i != argc; ++i) std::cout << " " << argv[i];
  std::cout << "\"\n";

  using namespace cphstl;
  using namespace pqfw_node;
  using namespace benchmarking;

```

```
80    string_filter output(argc>3?argv[3]:"rep");
81
82    int max_exp(5), intervals(3), r_min(11);
83    if(argc>4) {
84      std::stringstream ss(argv[4]); ss >> max_exp;
85    }
86    if(argc>5) {
87      std::stringstream ss(argv[5]); ss >> intervals;
88    }
89    if(argc>6) {
90      std::stringstream ss(argv[6]); ss >> r_min;
91    }
92    bool rep(output.pass("rep")), typ(output.pass("typ"))
93      , dat(output.pass("dat")), gnu(output.pass("gnu"))
94      , tst(output.pass("tst")), pro(output.pass("pro"));
95
96
97    typedef long long V;
98    //typedef int V;
99
100 #ifdef PROFILE
101   typedef std::less<V> C;
102   typedef std::allocator<V> A;
103   benchmark_main<profiling_counter, pq_benchmark, impl_translation
104   > bm(argc>1?argv[1]:"", argc>2?argv[2]:"");
105 #else
106   typedef counting_compare<std::less<V> > C;
107   typedef counting_allocator<> A;
108   benchmark_main<combi_counter, pq_benchmark, impl_translation
109   > bm(argc>1?argv[1]:"", argc>2?argv[2]:"");
110 #endif
111
112   std::cout << "# has_member: (M)eld, (C)lear, (P)op, (R)oot_list_type - friendly name and
          sizes.\n";
113
114 #ifdef INCL_LEDA
115   cphstl_leda::leda_heap<V,C,A>::registrate(bm);
116 #endif
117   vuillemin_alt1<counting_VCA<V> >::registrate(bm);
118 #ifdef INCL_PQFWS
119   pqfws<V,C,A>::registrate(bm);
120 #endif
121   vuillemin_alt2<counting_VCA<V> >::registrate(bm);
122 #ifndef NDEBUG
123   vuillemin<V,C,A>::registrate(bm); // exclude old variants if benchmarking
124 #endif
125
126   std::cout << "\n# datasize: " << (sizeof(V)*CHAR_BIT) << " bits\n";
127
128   if(rep || dat || tst || pro)
129     bm.run(max_exp, intervals, r_min, tst);
130
131   if(rep) bm.report();
132   if(typ) bm.type_report();
133   if(dat) bm.gnuplot_data();
134   if(gnu) bm.gnuplot_script(max_exp);
135
136   std::cout << "\n\n";
137
138   assertion_help(
139     if((count_yes+count_no) != 0) {
140       std::cout << "count_yes: " << count_yes
141         << ", count_no: " << count_no << ", yes fraction: "
142         << double(count_yes) / (count_yes+count_no) << "\n\n";
143     }
144   )
145
146   //system("pause");
147
148   leda::std_memory_mgr.clear();
149   return 0;
150 }
151
152
```

```
153 int main(int argc, char** argv) {
154 #ifdef debug_with_ddd
155   char* ddd[5] = {"", ".", "db_bno_r_dv", "rep", "5"}; // when debug gcc
156   return run(5,&ddd[0]);
157 #endif
158   std::cerr << "\n\n#-----␣run␣benchmarks␣-----\n";
159
160   if(argc<= 1) {
161     std::cerr << "arguments:␣[methods]␣[implementations]␣[output]␣[max_exp]␣[inter]␣[min_rep]\
          n"
162       << "where\n"
163       << "\tmethods:␣push,increase,erase,pop,meld,push_pop,worst_push,clear,stefan\n"
164       /* << "\tcounters: seconds,cycles,allocations,compares\n\n" */
165       << "\timplementations:␣db_bno_3_dv,ir_pwh_3_dv,..\n"
166       << "\toutput:␣rep(ort),typ(es),dat(a),gnu(plot),pro(file),tst\n\n"
167       << "\tmax_exp:␣the␣number␣of␣elements␣as␣2^max_exponent.\n"
168       << "\tinter:␣the␣number␣of␣intervals␣between␣every␣power␣of␣2.\n"
169       << "\t*)␣even␣distribution␣when␣inter␣is␣not␣a␣power␣of␣2:␣2^(k/inter).\n"
170       << "\t␣␣␣else␣uneven␣distribution:␣2^k␣+␣j*2^(1+k-log(inter)),␣j␣=␣0..inter-1.\n"
171       << "\tmin_rep:␣the␣minimum␣number␣of␣repetitions␣per␣element␣count.\n"
172       << "sample:␣'.␣db_bno_3_dv␣rep␣10␣3'␣(where␣'.'␣is␣the␣wildcard).\n";
173     return 1;
174   } else return run(argc,argv);
175 }
```

E Test

E.1 bit-store-dummy.cpp

```
#include "bit-store.h++"
```

E.2 encapsulator-node-test.i++

```
#ifndef _CPHSTL_PQFW_ENCAPSULATOR_NODE_TEST_IPP_
#define _CPHSTL_PQFW_ENCAPSULATOR_NODE_TEST_IPP_

/*
Desc: Test of weak-heap-node and its variants.
Auth: Jyrki Katajainen, Asger Bruun © 2009
Use:  Test your own node type, N, write the following:
        "encapsulator_node_test<N, char > > >::test();".
Warn: This test was accidentially written for std::greater,
      even though std::less is a more popular choice.
*/

#include "ms_c_fix.hpp"
#include "assert.h++"

#include <functional> // less
#include <iostream> // cout
#include <typeinfo> // typeid

#include "binomial_node.hpp"
#include "node_with_direct_value.hpp"
#include "node_with_facade.hpp"

namespace cphstl {
  namespace pqfw_node {

    template < typename E = w_facade<w_direct_value<binomial_node_base, char > >
//     , template <typename> class CT = std::greater
    >
    struct encapsulator_node_test { // test the node type given by E.
      typedef typename E::value_type V;
      typedef typename E::comparator_type C;
      //typedef CT<V> C;

      // Helpers.

      static int element_count(E* root) {
        int ret(0);
        for(E* r = root; r != 0; r = (*r).successor()) ++ret;
        return ret;
      }

      struct test_values {
        E a,b,c,d,e,f,g,h;
        test_values()
          : a('A'), b('B'), c('C'), d('D')
          , e('E'), f('F'), g('G'), h('H') {}

        E* build() {
          C cmp;
          return d.join(&b,cmp)->join(a.join(&c,cmp),cmp)
            ->join(h.join(&f,cmp)->join(e.join(&g,cmp),cmp),cmp);
        }
      };

      // The tests.

      static void execute(char const* friendly_type_name) {
        list(friendly_type_name);
        test_basic(friendly_type_name);
        test_root_find(friendly_type_name);
        test_promote(friendly_type_name);

```

```
64            if(node_traits<typename E::node_base_type>::has_splice == 0)
65              std::cout << "\nsplice not implemented and cannot be tested!\n";
66            else test_splice(friendly_type_name);
67
68            test_root_list(friendly_type_name);
69          }
70
71          static void test_pqfw(char const* friendly_type_name) { // obsolete??
72            list(friendly_type_name);
73            test_basic(friendly_type_name);
74            test_root_find(friendly_type_name);
75            test_promote(friendly_type_name);
76            test_splice(friendly_type_name);
77          }
78
79          static void list(char const* friendly_type_name) {
80            std::cout << friendly_type_name << " testing " << typeid_name_fix(typeid(E)) << "\n";
81            std::cout << "footprint: " << E::footprint() << "\n";
82          }
83
84          static void test_basic(char const* friendly_type_name) {
85            C cmp;
86            // check ascending first 4 values
87            //for(int i = 0; i != 3; i++) assert(!cmp(succs[i], succs[i+1]));
88
89            test_values tv;
90            std::cout << friendly_type_name << " setup: a = '" << tv.a.element() << "', b = '" <<
                  tv.b.element()
91              << "', c = '" << tv.c.element() << "', d = '" << tv.d.element() << "'.\n\n";
92
93            assert(tv.a.is_root());
94            assert(tv.a.tree_valid(cmp));
95            int n = element_count(&tv.a); assert(n == 1);
96
97            // join
98
99            std::cout << "p = d.join(b). ";
100           E* p = tv.d.join(&tv.b,cmp);
101           std::cout << "p: " << (*p).str() << "\n";
102           if (cmp(tv.d.element(), tv.b.element())) {
103             assert(p == &tv.b);
104             assert(tv.b.is_root());
105             assert(!tv.d.is_root());
106             assert(tv.d.distinguished_ancestor() == &tv.b);
107           } else {
108             assert(p == &tv.d);
109             assert(tv.d.is_root());
110             assert(!tv.b.is_root());
111             assert(tv.b.distinguished_ancestor() == &tv.d);
112           }
113           n = element_count(p); assert(n == 2);
114           assert((*p).tree_valid(cmp));
115
116           std::cout << "q = a.join(c). ";
117           E* q = tv.a.join(&tv.c,cmp);
118           std::cout << "q: " << (*q) << "\n";
119           if (cmp(tv.c.element(), tv.a.element())) {
120             assert(q == &tv.a);
121             assert(tv.a.is_root());
122             assert(!tv.c.is_root());
123           } else {
124             assert(q == &tv.c);
125             assert(tv.c.is_root());
126             assert(!tv.a.is_root());
127           }
128
129
130           assert(q != p);
131           assert((*q).tree_valid(cmp));
132
133           std::cout << "p = p.join(q). ";
134           p = (*p).join(q,cmp); std::cout << "p: " << (*p).str() << "\n\n";
135           assert(p == &tv.a);
136           assert(tv.a.is_root());
```

```
137
138           assert(tv.b.distinguished_ancestor() == &tv.a);
139           assert(tv.c.distinguished_ancestor() == &tv.a);
140           assert(tv.d.distinguished_ancestor() == &tv.b
141             || tv.d.distinguished_ancestor() == &tv.c);
142           assert((*p).tree_valid(cmp));
143
144           std::cout << "successors:";
145           for(E* r = p; r != 0; r = /*const_cast*/(*r).successor())
146             std::cout << " " << (*r).element();
147           std::cout << ".\n\n";
148           n = element_count(p); assert(n == 4);
149
150           std::cout << "q = p.split(). ";
151           q = (*p).split();
152           std::cout << "p: " << (*p).str() << " & q: " << (*q).str() << "\n";
153
154           assert((*p).tree_valid(cmp));
155           n = element_count(p); assert(n == 2);
156           assert((*q).tree_valid(cmp));
157           n = element_count(q); assert(n == 2);
158
159           std::cout << "p = p.join(q). ";
160           p = (*p).join(q,cmp);
161           std::cout << "p: " << (*p).str() << "\n\n";
162           assert(p == &tv.a);
163           n = element_count(p); assert(n == 4);
164           assert((*p).tree_valid(cmp));
165         }
166
167         static void test_root_list(char const* friendly_type_name) {
168           std::cout << friendly_type_name << " test root list\n";
169           test_values tt1;
170
171           E* p = tt1.build();
172           typename E::root_list_type l1(0);
173           E* t((*p).most_significant_child());
174           typename E::size_type n;
175           E* r((*p).construct(n, constant<bool,true>()));
176           //E* r((*p).construct(n));
177           l1.inject_range(r,t);
178
179           //test_values tt2;
180           //p = tt2.build();
181           //E::sized_root_list_type l2((*p).construct_with_size());
182
183           //std::cout << "sizeof(root_list_type), sizeof(sized_root_list_type): "
184           //  << sizeof(l1) << ", " << sizeof(l2);
185           //assert(sizeof(l1)+sizeof(void*) == sizeof(l2));
186           std::cout << "\n\n";
187         }
188
189         static void test_root_find(char const* friendly_type_name) {
190           std::cout << friendly_type_name << " test root find\n";
191           assertion_help(C cmp);
192           for(int t = 0; t != 8; ++t) {
193             test_values tt;
194             E* p = tt.build();
195             assert((*p).tree_valid(cmp));
196
197             E* r = p; // select element #t
198             for(int i = 0; i != t && r != 0; ++i) r = /*const_cast*/(*r).successor();
199             assert(r != 0); //analysis_assume(r != 0);
200
201             if((*r).is_root()) std::cout << "skip " << (*r).element() << " (is_root)\n\n";
202             else {
203               std::cout << "root found from " << (*r).element() << " in " << (*p).str() << "\n";
204               E* root = (*r).root();
205               assert(root == p);
206             }
207           }
208         }
209
210         struct dummy_heap_store {
```

```
211           typedef dummy_heap_store heap_proxy_type;
212           typedef dummy_heap_store encapsulator_type;
213
214           // !!! outdated void update(void* const) {;}
215           void replace(E*) {;}
216         };
217
218
219         static void test_splice(char const* friendly_type_name) {
220           std::cout << friendly_type_name << " test splice\n";
221           assertion_help(C cmp);
222           dummy_heap_store dummy;
223           typedef typename E::hole_type hole_type;
224           for(int t = 0; t != 8; ++t) {
225             test_values tt;
226             E* p = tt.build();
227             assert((*p).tree_valid(cmp));
228
229             E* r = p; // select element #t
230             for(int i = 0; i != t && r != 0; ++i) r = /*const_cast*/(*r).successor();
231             assert(r != 0); //analysis_assume(r != 0);
232
233             if((*r).is_root()) std::cout << "skip " << (*r).element() << " (is_root)\n\n";
234             else {
235               //std::cout << (*r).is_root();
236               std::cout << "splice_out " << (*r).element() << " from: " << (*p).str() << "\n";
237               hole_type h = (*r).splice_out();
238               std::cout << "r: " << (*r).str() << "\n";
239
240               (*r).splice_in(h, dummy);
241               std::cout << "splice_in , p: " << (*p).str() << ".\n\n";
242             }
243           }
244         }
245
246         static void test_promote(char const* friendly_type_name) {
247           std::cout << friendly_type_name << " test promote\n";
248           C cmp;
249           for(int t = 0; t != 8; ++t) {
250             test_values tt;
251             E* p = tt.build();
252             assert((*p).tree_valid(cmp));
253
254             E* r = p;
255             for(int i = 0; i != t && r != 0; ++i) r = /*const_cast*/(*r).successor();
256             assert(r != 0);
257
258             V v = (*p).element(); --v; // top-1
259
260             std::cout << "promote " << (*r).element();
261             (*r).element() = v;
262             std::cout << " to " << v << ": " << (*p).str() << "\n";
263             for(E* d = (*r).distinguished_ancestor();
264               d != 0 && !cmp(v, (*d).element());
265               d = (*r).distinguished_ancestor()
266               ) {
267                 (*r).promote(d);
268                 if((*r).is_root())
269                   std::cout << "r:                " << (*r).str() << "\n";
270                 else std::cout << "p:                " << (*p).str() << "\n";
271
272                 (*r).element() = (*d).element();
273                 assert((*r).tree_valid(cmp));
274                 (*r).element() = v;
275             }
276             assert((*r).is_root());
277             std::cout << "result:          " << (*r).str() << "\n\n";
278             assert((*r).distinguished_ancestor() == 0);
279             int n = element_count(r); assert(n == 8);
280           }
281         }
282       };
283     }
284   }
```

```
285
286 #if defined(ENCAPSULATOR_NODE_TEST)
287
288 #include <string>
289 #include <sstream> // ostringstream
290
291 int main(int, char**) {
292   using namespace cphstl::pqfw_node;
293   encapsulator_node_test<w_facade<w_direct_value<binomial_node_base, char> > >::execute();
294   return 0;
295 }
296
297 #endif
298 #endif
```

E.3 makefile.mk

```
1  # unit tests
2
3  # CPHSTL_ROOT=$(HOME)/CPHSTL
4  # CPHSTL_ROOT=/mnt/hgfs/cphstl
5  CPHSTL_ROOT=/media/win/cvs/CPHSTL
6  ASGER_ROOT=$(CPHSTL_ROOT)/Progress/Meldable-priority-queue/Asger-weak-binomial-mixed
7  CPHSTL_SOURCE=$(ASGER_ROOT)/CPHSTL_Branch
8
9  CXXFLAGS = -Wall -std=c++0x -x c++ -g
10 #CXXFLAGS = -Wall -std=c++0x -pedantic -x c++ -g
11 ## CXXFLAGS = -Wall -pedantic -ansi -x c++ -g
12 # IFLAGS = -I ../Code -I ../Benchmarking -I ../Code/CPHSTL_modified -I ../../../../Source
       /Common/Code -I ../../../../Source/Priority-queue-framework/Code -I ../../../../
       Source/Assert/Code -I ../../../../Source/Iterator/Code -I ../../../../Source/Meldable
       -priority-queue/Code -I ../../../../Source/Type/Code -I.
13 # IFLAGS = -I ../Code -I ../Benchmarking -I ../Code/CPHSTL_modified -I ../CPHSTL-Type -I
       ../CPHSTL-Iterator -I ../CPHSTL-Priority-queue-framework -I ../CPHSTL-Meldable-
       priority-queue -I ../../../../Source/Common/Code
14
15 IFLAGS = -I . -I ../Code -I ../Test -I ../Benchmarking -I ../Code/CPHSTL_modified -I $(
       CPHSTL_SOURCE)/Iterator/Code -I $(CPHSTL_SOURCE)/Common/Code -I $(CPHSTL_SOURCE)/Type/Code
        -I "$(CPHSTL_SOURCE)/Priority-queue-frameworks/Code" -I $(CPHSTL_SOURCE)/Proxy/Code -I $(
       CPHSTL_SOURCE)/Meldable-priority-queue/Code -I $(LEDAROOT)/incl -L$(LEDAROOT)
16
17 CXX = g++
18
19 unit_tests: pq_node_test pq_fw_test pq_benchmark
20
21 mini_test:
22   $(CXX) $(CXXFLAGS) $(IFLAGS) mini_test.cpp
23   ./a.out
24   @rm -f a.out
25
26 pq_node_test:
27   $(CXX) $(CXXFLAGS) $(IFLAGS) pq_node_test.cpp
28   ./a.out
29   @rm -f a.out
30
31 pq_fw_test:
32   $(CXX) $(CXXFLAGS) $(IFLAGS) pq_fw_test.cpp
33   ./a.out
34   @rm -f a.out
35
36 pq_benchmark:
37   $(CXX) $(CXXFLAGS) $(IFLAGS) ../Benchmark/pq_benchmark.cpp
38   ./a.out . !db_bno_r_dv rep 5
39   #./a.out . !db_bno_r_dv gnu 5 1> pq.gnu
40   #./a.out . !db_bno_r_dv dat 5 1> pq.dat
41   #@gnuplot pq.gnu
42 #fails  ./a.out . db_bno_r_dv rep 5
43 # @rm -f a.out
44
45
46
47 # framework-test:
48 #   $(CXX) $(CXXFLAGS) $(IFLAGS) -DUNITTEST_PRIORITY_QUEUE_FRAMEWORK ../Code/priority-
```

```
    queue-framework.h++
#   ./a.out > /dev/null
# # @rm -f a.out

# leak-check:
#   $(CXX) $(CXXFLAGS) $(IFLAGS) priority-queue-framework-test.c++
#   valgrind --leak-check=full --show-reachable=yes ./a.out

clean:
  @rm -vf *~ a.out core
```

E.4 make_test.cmd

```
cd /d %~p0

.\vs2008\Debug\pq_node_test.exe
if errorlevel 1 pause

.\vs2008\Debug\pq_fw_test.exe
if errorlevel 1 pause

.\vs2008\Debug\pq_benchmark.exe . !dl(l)_bno_3_dv,dl(l)_pwh_3_dv tst
if errorlevel 1 pause
```

E.5 mini_test.cpp

```
/*
Desc: Test unit_test_has_member and additional some copy constructor detection tests not
    ready for "has_member.hpp".
Auth: Asger Bruun, 2010.
Note: These tests were the first prototypes. I moved the experiments
      first into "direct_heap_store.hpp", "node_face.hpp" and a
      modified version of "meldable-priority-queue.h++" later on.
      I realize now after having choosen the final designs for automatic
      configuration on optional members that i forgot to repair the
      enable_if based test in "has_member.hpp".
*/

#include <iostream> // std::cout
#include <type_traits> // is_convertible

#define UNIT_TEST_HAS_MEMBER // void unit_test_has_member()
#include "has_member.hpp" // ...ENABLE_CUSTOM

#ifndef _WIN32_WINNT
#define _WIN32_WINNT 0x0600 // minimum required platform is Windows Vista.
#endif
#define WIN32_LEAN_AND_MEAN // exclude rare APIs such as Cryptography, DDE, RPC, Shell,
    and Windows Sockets.


template<typename T, typename S>
class ptr {
  // Interesting copy constructor example tested (in main) from: "C++0x: No concepts no
      fun?", http://pizer.wordpress.com/2009/08/16/c0x-no-concepts-no-fun/
  T* p;
  template<typename, typename> friend class ptr;

public:
  ptr(T* p) : p(p) {}

  template<typename U, typename V>
  ptr(ptr<U,V> const& x, typename std::enable_if<
    std::is_convertible<U*,T*>::value
    , void
  >::type* = 0)
  : p(x.p) {}

  T& operator*() const {return *p;}
  T* operator->() const {return p;}
  T* get() const {return p;}
};
```

```
44
45
46  DEFINE_STATIC_TEST_ENABLE_CUSTOM(cpy)
47
48  namespace cphstl {
49    template<typename V>
50    class rea_1 {
51    public:
52      static const bool custom_copy_construct = false;
53      static const bool enable_custom_cpy = false;
54      long long ll;
55      V v;
56      explicit rea_1() : v(0) { std::cout << "rea_1 empty \t"; }
57    private:
58      // "private" makes the MSC instric "__has_copy" to detect a false positive
59      // and makes GCC 4.4.1 fail compilation. therefore we cannot support:
60      //    rea_1(rea_1 const&); // undefined
61    };
62
63    template<typename V>
64    class rea_2 {
65    public:
66      static const bool custom_copy_construct = true;
67      static const bool enable_custom_cpy = true;
68      long long ll;
69      V v;
70      explicit rea_2() : v(0) { std::cout << "rea_2 empty \t"; }
71      explicit rea_2(rea_2 const& o) : v(o.v) { std::cout << "rea_2 copy \t"; }
72    };
73
74    template<typename V, typename R = rea_1<V> >
75    class bri {
76    public:
77      typedef bri<V,R> t_t;
78      typedef R r_t;
79      r_t r;
80      explicit bri() : r() { std::cout << "bri empty\n"; }
81
82      template<typename U>
83      bri(t_t const& o);
84
85      // ?: is it best to use "is_convertible" or "is_same" here? (GCC works with
            is_convertible only).
86      template<typename U>
87      explicit bri(U const& o, typename std::enable_if<
88        std::is_convertible<typename U::r_t*, R*>::value
89        && !enable_custom_cpy<R>::value
90        , void
91      >::type* p = 0) : r() { r.v = o.r.v; std::cout << "bri no copy\n"; }
92
93      template<typename U>
94      explicit bri(U const& o, typename std::enable_if<
95        std::is_convertible<typename U::r_t*, R*>::value
96        && enable_custom_cpy<R>::value
97        , void
98      >::type* p = 0) : r(o.r) { std::cout << "bri copy\n"; }
99    };
100 }
101
102 // -----
103
104 namespace cphstl {
105
106   class X{
107   public:
108     static const bool enable_custom_cpy = true;
109     typedef X copy_type; // this is a flag easily detected.
110     explicit X() {}
111     explicit X(X const& other) {}
112   };
113
114   class Y{
115   public:
116     explicit Y() {}
```

```
117     private:
118       explicit Y(Y const& other); // disabled
119     };
120
121     template <typename T>
122     struct has_copy_type {
123       typedef char yes;
124       typedef struct { char a[2]; } no;
125
126       template <typename U> static yes test(typename U::copy_type const*);
127       template <typename U> static no test(...);
128
129       static const bool value = sizeof(test<T>(0)) ==  sizeof(yes);
130     };
131
132     template <typename R>
133     class I {
134       template<typename> friend class I;
135       R r;
136     public:
137       //static const bool enable_custom_cpy = enable_custom_cpy<R>::value;
138       /*
139       set_custom_cpy
140       get_custom_cpy<R>::value
141       DEFINE_STATIC_TEST(enable_custom_cpy, false) // default
142       DEFINE_STATIC_TEST(disable_custom_cpy, true) // default
143       */
144       typedef R r_type;
145
146       explicit I() {}
147
148       template<typename T>
149       I( T& other,
150         typename std::enable_if<enable_custom_cpy<typename T::r_type>::value,void>::type* = 0
151         ) : r(other.r) { std::cout << "{custom_cpy}" << std::endl; }
152
153       template<typename T>
154       I( T& other,
155         typename std::enable_if<!enable_custom_cpy<typename T::r_type>::value,void>::type* = 0
156         ) : r() { std::cout << "{no␣custom_cpy}" << std::endl; }
157
158       //explicit I(I const& other) : r() { std::cout << "(nocpy)"; }
159     };
160   }
161
162   int main(int argc, char* argv[]) {
163     unit_test_has_member();
164
165     {
166       using namespace cphstl;
167       X x1, x2(x1);
168       Y y1 /* , forbidden y2(y1) because copy ctor disabled */;
169       I<X> ix1, ix2(ix1);
170       I<Y> iy1, iy2(iy1);
171
172       std::cout
173         << has_copy_type<X>::value
174         << "␣" << has_copy_type<Y>::value
175         << "␣" << has_copy_type<int>::value
176         << std::endl;
177
178     }
179
180     int i(7);
181     ptr<int,char> x(&i);
182     ptr<int,char> y(x);
183
184     using namespace cphstl;
185     typedef bri<int, rea_1<int> > bri_1;
186     typedef bri<int, rea_2<int> > bri_2;
187     bri_1 b1a;
188     bri_1 b1b(b1a);
189     bri_2 b2a;
190     bri_2 b2b(b2a);
```

```
191
192     std::enable_if<enable_custom_cpy<rea_2<int> >::value,void>::type* p = 0; p = 0;
193
194     std::cout << "\nstd::is_pod<rea_1<int> >::value " << std::is_pod<rea_1<int> >::value << "\n"
            ;
195     std::cout << "std::is_pod<rea_2<int> >::value " << std::is_pod<rea_2<int> >::value << "\n";
196 #ifdef _MSC_VER // experiemental doesnt work with GCC 4.4.1:
197     //std::cout << "\nhas_copy_constructor<rea_1<int> >::value " << has_copy_constructor<
            rea_1<int> >::value << "\n";
198     //std::cout << "has_copy_constructor<rea_2<int> >::value " << has_copy_constructor<
            rea_2<int> >::value << "\n";
199 #endif
200     std::cout << "\nenable_custum_cpy<rea_1<int> >::value " << enable_custom_cpy<rea_1<int> >::
            value << "\n";
201     std::cout << "enable_custom_cpy<rea_2<int> >::value " << enable_custom_cpy<rea_2<int> >::
            value << "\n";
202
203     system("pause");
204     return 0;
205 }
206 /*
207 MSVC9 output:
208
209     {custom_cpy}
210     {no custom_cpy}
211     1 0 0
212     rea_1 empty       bri empty
213     rea_1 empty       bri no copy
214     rea_2 empty       bri empty
215     rea_2 copy        bri copy
216
217     std::is_pod<rea_1<int> >::value 0
218     std::is_pod<rea_2<int> >::value 0
219
220     enable_custum_cpy<rea_1<int> >::value 0
221     enable_custom_cpy<rea_2<int> >::value 1
222
223
224 I cannot detect templated copy constructur safely in GCC,
225 GCC4.4.1 output:
226
227     {custom_cpy}
228     {no custom_cpy}
229     1 0 0
230     rea_1 empty     bri empty
231     // gcc bypasses this missing line because of auto copy ctor.
232     rea_2 empty     bri empty
233     rea_2 copy
234     std::is_pod<rea_1<int> >::value 0
235     std::is_pod<rea_2<int> >::value 0
236
237     enable_custum_cpy<rea_1<int> >::value 0
238     enable_custom_cpy<rea_2<int> >::value 1
239
240 */
```

E.6 pq_fw_test.cpp

```
1  #define INCLUDE_LEAK_CHECK
2
3  #include "ms_c_fix.hpp"
4  #include "assert.h++"
5
6  const bool careful(true);
7
8  #define DEBUG
9  #define UNITTEST_MULTIPLE_HEAP_FRAMEWORK
10 #ifdef _MSC_VER
11   #undef main
12 #endif
13 #define main pqfw_test_main
14 #include "multiple-heap-framework.h++"
15 #undef main
```

```
16
17 #include "direct_heap_store.hpp"
18 #include "heap_store_config.hpp"
19
20
21 #ifdef _MSC_VER
22   #define main checked_main // leak checked
23 #endif
24
25 #include "stl-meldable-priority-queue.h++"
26 #include "node-iterator.h++"
27
28 namespace cphstl {
29
30   template<typename T>
31   std::string bin_str(T v) {
32     const T base = 2;
33     std::string res;
34     do { res = "01"[v % base] + res; v /= base; } while(v);
35     return res;
36   }
37
38   template<typename T>
39   std::string red_bin_str(T c, T a) {
40     const T base = 2;
41     std::string res;
42     do { res = "012"[(c % base)+(a % base)] + res; c /= base; a /= base; } while(c|a);
43     return res;
44   }
45
46   void propa(redundant_binary_number_base<> & ccd) {
47     if(ccd.carry) {
48       redundant_binary_number_base<>::bit_scanner bs(ccd, 0);
49       while(!bs.is_carry()) bs.next_bit();
50       ccd.propagate(bs.current());
51     }
52   }
53   void inkr(redundant_binary_number_base<> & ccd, unsigned int i) {
54     while(i != 0) {
55       ccd.increment();
56       propa(ccd);
57       --i;
58     }
59   }
60
61   void number_system_demo() {
62     std::cout << "\nDemo the number system.\n";
63     {
64       redundant_binary_number_base<> ccd;
65       const unsigned int m = 50; //64+32;
66       std::vector<std::string> v(m+1);
67       for(unsigned int i = 0; i != (m+1); ++i) {
68         //std::cout << "  ccd(" << i << "): "
69         //  << bin_str(ccd.carry) << ", " << bin_str(ccd.accu)
70         //  << " -> " << red_bin_str(ccd.carry, ccd.accu) << "\n";
71         assert(ccd.value() == i);
72         v[i] = red_bin_str(ccd.carry, ccd.accu);
73         ccd.increment();
74         propa(ccd);
75       }
76
77       const unsigned int n = m/10;
78       for(unsigned int j = 0; j != n; ++j)
79         for(unsigned int i = 0; i != m; i+= n) {
80           std::cout << std::setw(6) << v[i+j+1];
81           std::cout << ((i == (m-n))?" \\\\\n":" & ");
82         }
83         std::cout << "\n\n";
84     }
85     {
86       redundant_binary_number_base<> ccd;
87       const unsigned int m = 50; //64+32;
88       std::vector<std::string> v(m+1);
89       for(unsigned int i = 0; i != (m+1); ++i) {
```

```
90          //std::cout << "  ccd(" << i << "): "
91          //  << bin_str(ccd.carry) << ", " << bin_str(ccd.accu)
92          //  << " -> " << red_bin_str(ccd.carry, ccd.accu) << "\n";
93          assert(ccd.value() == i);
94          v[i] = red_bin_str(ccd.carry, ccd.accu);
95          propa(ccd);
96          ccd.increment();
97        }
98
99        const unsigned int n = m/10;
100       for(unsigned int j = 0; j != n; ++j)
101         for(unsigned int i = 0; i != m; i+= n) {
102           std::cout << std::setw(6) << v[i+j+1];
103           std::cout << ((i == (m-n))?"␣\\\\␣\n":"␣&␣");
104         }
105         std::cout << "\n\n";
106     }
107
108     redundant_binary_number_base<> five, twenty;
109     inkr(five,5);
110     inkr(twenty,20);
111     std::cout << "five:␣" << red_bin_str(five.carry, five.accu)<< "\n";
112     std::cout << "twenty:␣" << red_bin_str(twenty.carry, twenty.accu)<< "\n";
113     std::cout << "twentyfive␣a:␣" << red_bin_str(twenty.carry, twenty.accu+five.carry+five.
          accu)<< "\n";
114     std::cout << "twentyfive␣b:␣" << red_bin_str(five.carry, five.accu+twenty.carry+twenty.
          accu)<< "\n";
115     std::cout << "\n\n";
116   }
117
118   void number_system_test() {
119     std::cout << "\nTest␣the␣number␣system.\n";
120     redundant_binary_number_base<> ccd;
121
122     for(unsigned int i = 0; i != 16; ++i) {
123       std::cout << "␣␣ccd(" << i << "):␣"
124         << bin_str(ccd.carry) << ",␣" << bin_str(ccd.accu)
125         << "␣->␣" << red_bin_str(ccd.carry, ccd.accu) << "\n";
126       assert(ccd.value() == i);
127       if(ccd.carry) {
128         redundant_binary_number_base<>::bit_scanner bs(ccd, 0);
129         //std::cout << "  bit_scanner: " << bs << "\n";
130         while(!bs.is_carry()) bs.next_bit();
131         ccd.propagate(bs.current());
132         std::cout << "␣␣bit_scanner:␣" << redundant_binary_number_base<>::bit_scanner(ccd, 0)
              << "\n";
133       }
134       ccd.increment();
135       std::cout << "␣␣bit_scanner:␣" << redundant_binary_number_base<>::bit_scanner(ccd, 0) <<
            "\n";
136     }
137     for(unsigned int i = 16; i != 0; --i) {
138       assert(ccd.value() == i);
139       ccd.decrement();
140       std::cout << "␣␣bit_scanner:␣" << redundant_binary_number_base<>::bit_scanner(ccd, 0) <<
            "\n";
141     }
142   }
143
144   namespace pqfw_node {
145
146     template <typename Q>
147     void test() {
148       pqfw_test<Q>::small_test_multiple_heap_framework();
149       pqfw_test<Q>::big_test_multiple_heap_framework();
150     }
151
152     template <typename E>
153     struct pqfw_realisator_test {
154       static void execute(char const* friendly_type_name) {
155         std::cout << "\n\nEncapsulator␣'" << friendly_type_name
156           << "':␣" << typeid_name_fix(typeid(E)) << "";
157
158         std::cout << "\nTest␣direct␣binay";
```

```
159           test<typename hs_db_slow_top<E>::R>();
160
161           std::cout << "\nTest indirect redundant binay";
162           if(node_traits<typename E::node_base_type>::has_splice == 0)
163             std::cout << "\n  skipped because splice not implemented\n";
164           else if(node_traits<typename E::node_base_type>::has_owner == 0)
165             std::cout << "\n  skipped because owner not implemented\n";
166           else test<typename hs_ir<E>::R>();
167         }
168       };
169     } // namespace pqfw_node
170   } //namespace cphstl
171
172
173   void pq_fw_test() {
174     using namespace cphstl;
175     using namespace pqfw_node;
176
177     number_system_test();
178
179     std::cout << "\n\n----- test priority queues\n";
180
181     typedef int V;
182     typedef std::less<V> C;
183     typedef std::allocator<V> A;
184     typedef node_config<V,C,A> nc;
185
186     nc::execute<pqfw_realisator_test>(); // test all node types
187
188     std::cout << "\nTest the slow top.";
189     test<hs_db_slow_top<nc::bno_3_dv>::R>();
190
191     std::cout << "\nTest the fast top wrapper.";
192     test<hs_db<nc::bno_3_dv>::R>();
193
194     std::cout << "\nTest the Cormen extract.";
195     test<hs_db<nc::bno_3_dv>::R_with_cormen_extract >();
196
197     std::cout << "\nTest the CPH STL extract.";
198     test<hs_db<nc::bno_3_dv>::R_with_cphstl_extract >();
199
200     /* // does not work with gcc:
201    // typedef js_policy<join_schedule_ultralight> jsu;
202    // typedef js_policy<join_schedule_light> jsl;
203    // typedef js_policy<join_schedule_medium> jsm;
204    // typedef js_policy<join_schedule_fat> jsf;
205
206    // std::cout << "\nTest the redundant binary number (ultralight schedule).";
207    // test<hs_drbc<nc::bno_3_dv, jsu>::R>();
208
209    // std::cout << "\nTest the redundant binary number (light schedule).";
210    // test<hs_drbc<nc::bno_3_dv, jsl>::R>();
211
212     //std::cout << "\nTest the redundant binary number (medium schedule).";
213    // test<hs_drbc<nc::bno_3_dv, jsm>::R>();
214
215     //std::cout << "\nTest the redundant binary number (fat schedule).";
216    // test<hs_drbc<nc::bno_3_dv, jsf>::R>();
217   */
218
219     std::cout << "\nTest the redundant binary number (ultralight schedule).";
220     test<hs_drb<nc::bno_3_dv>::R_js_u>();
221
222     std::cout << "\nTest the redundant binary number (light schedule).";
223     test<hs_drb<nc::bno_3_dv>::R_js_l>();
224
225     std::cout << "\nTest the redundant binary number (medium schedule).";
226     test<hs_drb<nc::bno_3_dv>::R_js_m>();
227
228     std::cout << "\nTest the redundant binary number (fat schedule).";
229     test<hs_drb<nc::bno_3_dv>::R_js_f>();
230
231
232
```

```
233     /* // old style was:
234        typedef with_fast_top<direct_heap_store<w_facade<
235         w_direct_value<binomial_node_base, V, C, A> > >
236        > pq_db_bno_3_dv; // std binomial queue
237        typedef multiple_heap_framework<V, C, A, w_facade<
238          w_direct_value<pwh_node_base, V, C, A> >
239        > pq_ir_pwh_3_dv; // std weak queue
240        test<pq_db_bno_3_dv>();
241        test<pq_ir_pwh_3_dv>();
242     */
243
244
245     std::cout << "\n\n------ test priority queues\n";
246   }
247
248   int main(int /*argc*/, char** /*argv*/) {
249     using namespace cphstl;
250     using namespace pqfw_node;
251
252     //std::cout.rdbuf(0); // make cout silent.
253     pq_fw_test();
254     number_system_demo();
255     system_pause();
256     return 0;
257   }
258
259   #ifdef NDEBUG
260   #error the test is useles without assertions enabled.
261   #endif
```

E.7 pq_node_test.cpp

```
1   #define INCLUDE_LEAK_CHECK
2
3   #include "ms_c_fix.hpp"
4   #include "assert.h++"
5
6   const bool careful(false);
7
8   #include "encapsulator-node-test.i++"
9   #include "node_config.hpp"
10
11  namespace cphstl {
12    namespace pqfw_node {
13
14      void pq_node_test() {
15        std::cout << "\n\n------ test node types\n";
16
17        typedef node_config<char> nc;
18
19        nc::execute<encapsulator_node_test>();
20      }
21
22      void pq_node_sizeof_test() {
23        // Desc: protect yourself against accidential data members.
24        std::cout << "\n\n------ test size of nodes\n";
25        assert(sizeof(binomial_node_base) == 3*sizeof(void*));
26        assert(sizeof(pwh_node_base) == 3*sizeof(void*));
27        assert(sizeof(light_binomial_node_base) == 2*sizeof(void*));
28        assert(sizeof(brown_r_node<long>) == 2*sizeof(void*)+sizeof(long));
29        assert(sizeof(brown_k_node<long>) == 2*sizeof(void*)+sizeof(long));
30
31        assert(sizeof(w_direct_value<pwh_node_base,int>) == sizeof(pwh_node_base)+sizeof(int)
              );
32        assert(sizeof(w_direct_value<pwh_node_base,int>::value_type) == sizeof(int));
33        assert(sizeof(w_indirect_value<pwh_node_base,long>) == sizeof(void*)+sizeof(long));
34        assert(sizeof(w_facade<w_direct_value<pwh_node_base,int> >)
35          == sizeof(w_direct_value<pwh_node_base,int>));
36
37        assert(sizeof(binomial_node_base) == binomial_node_base::footprint());
38        assert(sizeof(pwh_node_base) == pwh_node_base::footprint());
39        assert(sizeof(light_binomial_node_base) == light_binomial_node_base::footprint());
40        assert(sizeof(brown_r_node<long>) == brown_r_node<long>::footprint());
```

```
41          assert(sizeof(brown_k_node<long>) == brown_k_node<long>::footprint());
42
43          /*
44          // why wont this compile???:
45          assert(w_facade<w_indirect_value<pwh_node_base,int> >::footprint() == 0);
46          assert(w_indirect_value<pwh_node_base,int>::footprint() == 0);
47
48          assert(
49            w_indirect_value<pwh_node_base,int>::footprint()
50            ==
51            (sizeof(w_indirect_value<pwh_node_base,int>)
52            +sizeof(w_direct_value<pwh_node_base,int>))
53          );
54          */
55        }
56      }
57    }
58    int main(int /*argc*/, char** /*argv*/) {
59      using namespace cphstl;
60      using namespace pqfw_node;
61
62      //std::cout.rdbuf(0); // make std::cout silent.
63
64      pq_node_sizeof_test();
65      pq_node_test();
66      std::cout << "\n\n------ pq_node_test\n";
67      system_pause();
68      return 0;
69    }
70
71    #ifdef NDEBUG
72    #error the test is useles without assertions enabled.
73    #endif
```

F CPHSTL_Branch/Iterator/Code

F.1 bidirectional-monolith-iterator.h++

```
1  /*
2    The idea of combining iterators and const iterators into the same
3    class is taken from [Matt Austern. Defining iterators and const
4    iterators. C/C++ User's Journal 19,1 (2001), 74-79].
5
6    Authors: Jyrki Katajainen, Bo Simonsen, 2006, 2008
7  */
8
9  #ifndef _CPHSTL_NODE_ITERATOR_
10 #define _CPHSTL_NODE_ITERATOR_
11
12 #include <cstddef> // std::ptrdiff_t
13 #include <iostream> // std::ostream
14 #include <iterator> // std::bidirectional_iterator_tag
15 #include <string> // std::string
16 #include "type.h++" // cphstl::if_then_else
17
18 namespace leda {
19   template < typename K, typename I, typename C, typename R >
20   class dictionary;
21 }
22
23 namespace cphstl {
24
25   /* Forward declarations of container classes */
26
27   template <typename V, typename A, typename R, typename I, typename J>
28   class list;
29
30   template <typename V, typename C, typename A, typename R,
31       typename I, typename J>
32   class multiset;
33
34   template <typename V, typename C, typename A, typename R,
35       typename I, typename J>
36   class set;
37
38   template <typename K, typename V, typename C, typename A, typename R,
39       typename I, typename J>
40   class set_base;
41
42   template <typename K, typename V, typename C, typename A, typename R,
43       typename I, typename J>
44   class map;
45
46   template <typename K, typename V, typename C, typename A, typename R,
47       typename I, typename J>
48   class map_base;
49
50
51   template <typename V, typename C, typename A, typename E,
52     typename R, typename I, typename J>
53   class meldable_priority_queue;
54
55
56   template <typename N, bool is_const = false>
57   class node_iterator {
58
59   public:
60
61     // types
62
63     typedef std::bidirectional_iterator_tag iterator_category;
64     typedef typename N::value_type value_type;
65     typedef std::ptrdiff_t difference_type;
66
67     typedef typename if_then_else<is_const, value_type const*, value_type*>::type pointer;
68     typedef typename if_then_else<is_const, value_type const&, value_type&>::type reference;
69
```

```
70      typedef node_iterator<N, !is_const> complement;
71    private:
72
73      // types
74
75      typedef typename if_then_else<is_const, N const*, N*>::type node_pointer;
76
77    public:
78
79      // friends
80      friend class node_iterator<N, !is_const>;
81
82      template <typename M, bool both>
83      friend std::ostream& operator<<(std::ostream&, node_iterator<M, both> const&);
84
85      template <typename V, typename A, typename R, typename I, typename J>
86      friend class cphstl::list;
87
88      template <typename V, typename C, typename A, typename R,
89          typename I, typename J>
90      friend class cphstl::set;
91
92      template <typename K, typename V, typename C, typename A, typename R,
93          typename I, typename J>
94      friend class cphstl::set_base;
95
96      template <typename V, typename C, typename A, typename R,
97          typename I, typename J>
98      friend class cphstl::multiset;
99
100     template <typename K, typename V, typename C, typename A,
101         typename R, typename I, typename J>
102     friend class cphstl::map;
103
104     template <typename K, typename V, typename C, typename A, typename R,
105         typename I, typename J>
106     friend class cphstl::map_base;
107
108
109     template <typename V, typename C, typename A, typename E,
110       typename R, typename I, typename J>
111       friend class cphstl::meldable_priority_queue;
112
113
114     template <typename T1, typename T2>
115     friend class std::pair;
116
117     template < typename K, typename I, typename C, typename R >
118     friend class leda::dictionary;
119
120     // structors
121
122     node_iterator();
123     node_iterator(node_iterator<N, false> const&);
124     template<typename R>
125     node_iterator(node_iterator<N, false> const&, R const&);
126     node_iterator& operator = (node_iterator const&);
127     ~node_iterator();
128
129     // operators
130
131     reference operator*() const;
132     pointer operator->() const;
133     pointer operator->();
134     node_iterator& operator++();
135     node_iterator operator++(int);
136     node_iterator& operator--();
137     node_iterator operator--(int);
138
139     template <bool both>
140     bool operator == (node_iterator<N, both> const&) const;
141
142     template <bool both>
143     bool operator != (node_iterator<N, both> const&) const;
```

```
144
145   private:
146     // converters to be used by the friends
147
148     node_iterator(node_pointer); // node_pointer --> iterator
149     operator node_pointer() const; // iterator --> node_pointer
150     operator std::string() const; // iterator --> string
151
152     node_pointer link;
153
154   };
155
156 }
157
158 #include "node-iterator.i++" // implements cphstl::node_iterator
159
160 #endif
```

F.2 bidirectional-monolith-iterator.i++

```
1   /*
2     Implementation of cphstl::node_iterator
3
4     Author: Jyrki Katajainen, Bo Simonsen, April 2008
5
6     General design idea is proposed by J. Katajainen and B. Simonsen
7   */
8
9   #include <sstream> // defines std::stringstream
10
11  namespace cphstl {
12
13    // default constructor
14
15    template <typename N, bool is_const>
16    node_iterator<N, is_const>::node_iterator()
17      : link(0) {
18    }
19
20    // copy constructor
21
22    template <typename N, bool is_const>
23    node_iterator<N, is_const>::node_iterator(node_iterator<N, false> const& a)
24      : link(a.link) {
25    }
26    template <typename N, bool is_const>
27    template<typename R>
28    node_iterator<N, is_const>::node_iterator(node_iterator<N, false> const& a, R const&)
29      : link(a.link) {
30    }
31
32
33    // assignment
34
35    template <typename N, bool is_const>
36    node_iterator<N, is_const>&
37    node_iterator<N, is_const>::operator = (node_iterator<N, is_const> const& a) {
38      link = a.link;
39      return *this;
40    }
41
42    // destructor
43
44    template <typename N, bool is_const>
45    node_iterator<N, is_const>::~node_iterator() {
46    }
47
48    // operator*
49
50    template <typename N, bool is_const>
51    typename node_iterator<N, is_const>::reference
52    node_iterator<N, is_const>::operator*() const {
53      return reference((*link).content());
```

```
54      }
55
56      // operator->
57
58      template <typename N, bool is_const>
59      typename node_iterator<N, is_const>::pointer
60      node_iterator<N, is_const>::operator->() const {
61        return pointer(&(*link).content());
62      }
63
64      template <typename N, bool is_const>
65      typename node_iterator<N, is_const>::pointer
66      node_iterator<N, is_const>::operator->() {
67        return pointer(&(*link).content());
68      }
69
70      // operator++; pre-increment
71
72      template <typename N, bool is_const>
73      node_iterator<N, is_const>&
74      node_iterator<N, is_const>::operator++() {
75        link = (*link).successor();
76        return *this;
77      }
78
79       // operator++; post-increment
80
81      template <typename N, bool is_const>
82      node_iterator<N, is_const>
83      node_iterator<N, is_const>::operator++(int) {
84        node_iterator<N, is_const> temporary(*this);
85        ++(*this);
86        return temporary;
87      }
88
89      // operator--; pre-increment
90
91      template <typename N, bool is_const>
92      node_iterator<N, is_const>&
93      node_iterator<N, is_const>::operator--() {
94        link = (*link).predecessor();
95        return *this;
96      }
97
98      // operator--; post-increment
99
100     template <typename N, bool is_const>
101     node_iterator<N, is_const>
102     node_iterator<N, is_const>::operator--(int) {
103       node_iterator<N, is_const> temporary(*this);
104       --(*this);
105       return temporary;
106     }
107
108     // operator ==
109
110     template <typename N, bool is_const>
111     template <bool both>
112     bool
113     node_iterator<N, is_const>::operator == (node_iterator<N, both> const& a) const {
114       return link == a.link;
115     }
116
117     // operator !=
118
119     template <typename N, bool is_const>
120     template <bool both>
121     bool
122     node_iterator<N, is_const>::operator != (node_iterator<N, both> const& a) const {
123       return link != a.link;
124     }
125
126     // parametrized constructor (node -> iterator)
127
```

```
128   template <typename N, bool both>
129   node_iterator<N, both>::node_iterator(node_pointer p)
130     : link(p) {
131   }
132
133   // conversion operators (iterator -> node)
134
135   template <typename N, bool is_const>
136   node_iterator<N, is_const>::operator node_pointer() const {
137     return link;
138   }
139
140   // conversion operator (makes it possible to print out an iterator)
141
142   template <typename N, bool is_const>
143   node_iterator<N, is_const>::operator std::string() const {
144     std::stringstream ss;
145     std::string address;
146     ss << (int)(char*)((*this).link);
147     ss >> address;
148
149     if (is_const == false) {
150       return std::string("iterator: node at ") + address;
151     }
152     else {
153       return std::string("const_iterator: node at ") + address;
154     }
155   }
156
157   // representation
158
159   template <typename N, bool both>
160   std::ostream&
161   operator<<(std::ostream& s, node_iterator<N, both> const& i) {
162     s  << std::string(i);
163     return s;
164   }
165
166 }
```

F.3 node-iterator.h++

```
1  /*
2    The idea of combining iterators and const iterators into the same
3    class is taken from [Matt Austern. Defining iterators and const
4    iterators. C/C++ User's Journal 19,1 (2001), 74-79].
5
6    Authors: Jyrki Katajainen, Bo Simonsen, 2006, 2008
7  */
8
9  #ifndef _CPHSTL_NODE_ITERATOR_
10 #define _CPHSTL_NODE_ITERATOR_
11
12 #include <cstddef> // std::ptrdiff_t
13 #include <iostream> // std::ostream
14 #include <iterator> // std::bidirectional_iterator_tag
15 #include <string> // std::string
16 #include "type.h++" // cphstl::if_then_else
17
18 namespace leda {
19   template < typename K, typename I, typename C, typename R >
20   class dictionary;
21 }
22
23 namespace cphstl {
24
25   /* Forward declarations of container classes */
26
27   template <typename V, typename A, typename R, typename I, typename J>
28   class list;
29
30   template <typename V, typename C, typename A, typename R,
31       typename I, typename J>
```

```
32      class multiset;
33
34      template <typename V, typename C, typename A, typename R,
35          typename I, typename J>
36      class set;
37
38      template <typename K, typename V, typename C, typename A, typename R,
39          typename I, typename J>
40      class set_base;
41
42      template <typename K, typename V, typename C, typename A, typename R,
43          typename I, typename J>
44      class map;
45
46      template <typename K, typename V, typename C, typename A, typename R,
47          typename I, typename J>
48      class map_base;
49
50
51      template <typename V, typename C, typename A, typename E,
52        typename R, typename I, typename J>
53      class meldable_priority_queue;
54
55      template <typename CFG>
56      class meldable_priority_queue_alt;
57
58      template <typename N, bool is_const = false>
59      class node_iterator {
60
61      public:
62
63        // types
64
65        typedef std::bidirectional_iterator_tag iterator_category;
66        typedef typename N::value_type value_type;
67        typedef std::ptrdiff_t difference_type;
68
69        typedef typename if_then_else<is_const, value_type const*, value_type*>::type pointer;
70        typedef typename if_then_else<is_const, value_type const&, value_type&>::type reference;
71
72        typedef node_iterator<N, !is_const> complement;
73      private:
74
75        // types
76
77        typedef typename if_then_else<is_const, N const*, N*>::type node_pointer;
78
79      public:
80
81        // friends
82        friend class node_iterator<N, !is_const>;
83
84        template <typename M, bool both>
85        friend std::ostream& operator<<(std::ostream&, node_iterator<M, both> const&);
86
87        template <typename V, typename A, typename R, typename I, typename J>
88        friend class cphstl::list;
89
90        template <typename V, typename C, typename A, typename R,
91            typename I, typename J>
92        friend class cphstl::set;
93
94        template <typename K, typename V, typename C, typename A, typename R,
95            typename I, typename J>
96        friend class cphstl::set_base;
97
98        template <typename V, typename C, typename A, typename R,
99            typename I, typename J>
100       friend class cphstl::multiset;
101
102       template <typename K, typename V, typename C, typename A,
103           typename R, typename I, typename J>
104       friend class cphstl::map;
105
```

```
106       template <typename K, typename V, typename C, typename A, typename R,
107           typename I, typename J>
108       friend class cphstl::map_base;
109
110
111       template <typename V, typename C, typename A, typename E,
112         typename R, typename I, typename J>
113         friend class cphstl::meldable_priority_queue;
114
115       template <typename CFG>
116         friend class cphstl::meldable_priority_queue_alt;
117
118
119       template <typename T1, typename T2>
120       friend class std::pair;
121
122       template < typename K, typename I, typename C, typename R >
123       friend class leda::dictionary;
124
125       // structors
126
127       node_iterator();
128       node_iterator(node_iterator<N, false> const&);
129       template<typename R>
130       node_iterator(node_iterator<N, false> const&, R const&);
131       node_iterator& operator = (node_iterator const&);
132       ~node_iterator();
133
134       // operators
135
136       reference operator*() const;
137       pointer operator->() const;
138       pointer operator->();
139       node_iterator& operator++();
140       node_iterator operator++(int);
141       node_iterator& operator--();
142       node_iterator operator--(int);
143
144       template <bool both>
145       bool operator == (node_iterator<N, both> const&) const;
146
147       template <bool both>
148       bool operator != (node_iterator<N, both> const&) const;
149
150     private:
151       // converters to be used by the friends
152
153       node_iterator(node_pointer); // node_pointer --> iterator
154       operator node_pointer() const; // iterator --> node_pointer
155       operator std::string() const; // iterator --> string
156
157       node_pointer link;
158
159     };
160
161   }
162
163   #include "node-iterator.i++" // implements cphstl::node_iterator
164
165   #endif
```

F.4 node-iterator.i++

```
1   /*
2     Implementation of cphstl::node_iterator
3
4     Author: Jyrki Katajainen, Bo Simonsen, April 2008
5
6     General design idea is proposed by J. Katajainen and B. Simonsen
7   */
8
9   #include <sstream> // defines std::stringstream
10
```

```
11  namespace cphstl {
12
13    // default constructor
14
15    template <typename N, bool is_const>
16    node_iterator<N, is_const>::node_iterator()
17      : link(0) {
18    }
19
20    // copy constructor
21
22    template <typename N, bool is_const>
23    node_iterator<N, is_const>::node_iterator(node_iterator<N, false> const& a)
24      : link(a.link) {
25    }
26    template <typename N, bool is_const>
27    template<typename R>
28    node_iterator<N, is_const>::node_iterator(node_iterator<N, false> const& a, R const&)
29      : link(a.link) {
30    }
31
32
33    // assignment
34
35    template <typename N, bool is_const>
36    node_iterator<N, is_const>&
37    node_iterator<N, is_const>::operator = (node_iterator<N, is_const> const& a) {
38      link = a.link;
39      return *this;
40    }
41
42    // destructor
43
44    template <typename N, bool is_const>
45    node_iterator<N, is_const>::~node_iterator() {
46    }
47
48    // operator*
49
50    template <typename N, bool is_const>
51    typename node_iterator<N, is_const>::reference
52    node_iterator<N, is_const>::operator*() const {
53      return reference((*link).content());
54    }
55
56    // operator->
57
58    template <typename N, bool is_const>
59    typename node_iterator<N, is_const>::pointer
60    node_iterator<N, is_const>::operator->() const {
61      return pointer(&(*link).content());
62    }
63
64    template <typename N, bool is_const>
65    typename node_iterator<N, is_const>::pointer
66    node_iterator<N, is_const>::operator->() {
67      return pointer(&(*link).content());
68    }
69
70    // operator++; pre-increment
71
72    template <typename N, bool is_const>
73    node_iterator<N, is_const>&
74    node_iterator<N, is_const>::operator++() {
75      link = (*link).successor();
76      return *this;
77    }
78
79     // operator++; post-increment
80
81    template <typename N, bool is_const>
82    node_iterator<N, is_const>
83    node_iterator<N, is_const>::operator++(int) {
84      node_iterator<N, is_const> temporary(*this);
```

```
85      ++(*this);
86      return temporary;
87    }
88
89    // operator--; pre-increment
90
91    template <typename N, bool is_const>
92    node_iterator<N, is_const>&
93    node_iterator<N, is_const>::operator--() {
94      link = (*link).predecessor();
95      return *this;
96    }
97
98    // operator--; post-increment
99
100   template <typename N, bool is_const>
101   node_iterator<N, is_const>
102   node_iterator<N, is_const>::operator--(int) {
103     node_iterator<N, is_const> temporary(*this);
104     --(*this);
105     return temporary;
106   }
107
108   // operator ==
109
110   template <typename N, bool is_const>
111   template <bool both>
112   bool
113   node_iterator<N, is_const>::operator == (node_iterator<N, both> const& a) const {
114     return link == a.link;
115   }
116
117   // operator !=
118
119   template <typename N, bool is_const>
120   template <bool both>
121   bool
122   node_iterator<N, is_const>::operator != (node_iterator<N, both> const& a) const {
123     return link != a.link;
124   }
125
126   // parametrized constructor (node -> iterator)
127
128   template <typename N, bool both>
129   node_iterator<N, both>::node_iterator(node_pointer p)
130     : link(p) {
131   }
132
133   // conversion operators (iterator -> node)
134
135   template <typename N, bool is_const>
136   node_iterator<N, is_const>::operator node_pointer() const {
137     return link;
138   }
139
140   // conversion operator (makes it possible to print out an iterator)
141
142   template <typename N, bool is_const>
143   node_iterator<N, is_const>::operator std::string() const {
144     std::stringstream ss;
145     std::string address;
146     ss << (int)(char*)((*this).link);
147     ss >> address;
148
149     if (is_const == false) {
150       return std::string("iterator: node at ") + address;
151     }
152     else {
153       return std::string("const_iterator: node at ") + address;
154     }
155   }
156
157   // representation
158
```

```
159   template <typename N, bool both>
160   std::ostream&
161   operator<<(std::ostream& s, node_iterator<N, both> const& i) {
162     s << std::string(i);
163     return s;
164   }
165
166 }
```

F.5 priority-queue-iterator.h++

```
1  /*
2    An iterator to be used in our priority-queue framework. Each
3    priority queue is a queue of perfect components. An iterator holds a
4    pointer to a node. To access the next node, we call the successor
5    for the given node; if the node has no successor in its current
6    component, the first node (if any) in the following component is
7    returned.
8
9    Observe that for meldable structures only unidirectional iterators
10   can be supported! Therefore, operator-- is not provided.
11
12   Author: Jyrki Katajainen Â© 2009
13 */
14
15 #ifndef _CPHSTL_PRIRIOTY_QUEUE_ITERATOR_
16 #define _CPHSTL_PRIRIOTY_QUEUE_ITERATOR_
17
18 #include <cstddef> // std::ptrdiff_t
19 #include <iostream> // std::ostream
20 #include <iterator> // std::forward_iterator_tag
21 #include <string> // std::string
22
23 namespace {
24
25   // if statement for compile-time meta-programming
26
27   template <bool, typename T, typename U>
28   class if_then_else;
29
30   template <typename T, typename U>
31   class if_then_else<true, T, U> {
32   public:
33     typedef T type;
34   };
35
36   template <typename T, typename U>
37   class if_then_else<false, T, U> {
38   public:
39     typedef U type;
40   };
41 }
42
43 namespace cphstl {
44
45   template <typename V, typename C, typename A, typename F,
46       typename R, typename I, typename J>
47   class meldable_priority_queue;
48
49   template <typename CFG>
50   class meldable_priority_queue_alt;
51
52   template <typename E, typename R, bool is_const = false>
53   class priority_queue_iterator {
54   public:
55
56     // types
57
58     typedef std::forward_iterator_tag iterator_category;
59     typedef E encapsulator_type;
60     typedef R realizator_type;
61     typedef typename E::value_type value_type;
62     typedef std::ptrdiff_t difference_type;
```

```
63
64      typedef typename if_then_else<is_const , value_type const*, value_type*>::type pointer;
65      typedef typename if_then_else<is_const , value_type const&, value_type&>::type reference;
66
67      typedef priority_queue_iterator<E, R, !is_const> complement;
68
69    private:
70
71      typedef typename if_then_else<is_const , E const*, E*>::type node_pointer;
72
73    public:
74
75      // friends
76
77      friend class priority_queue_iterator<E, R, !is_const>;
78
79      template <typename F, typename Q, bool both>
80      friend std::ostream& operator<<(std::ostream&, priority_queue_iterator<F, Q, both> const
            &);
81
82      template <typename V, typename C, typename A, typename F, typename S,
83          typename I, typename J>
84      friend class cphstl::meldable_priority_queue;
85
86      template <typename CFG>
87      friend class cphstl::meldable_priority_queue_alt;
88
89      // structors
90
91      priority_queue_iterator();
92      priority_queue_iterator(priority_queue_iterator<E, R, false> const&);
93      priority_queue_iterator& operator = (priority_queue_iterator const&);
94      ~priority_queue_iterator();
95
96      // operators
97
98      reference operator*() const;
99      pointer operator->() const;
100     pointer operator->();
101     priority_queue_iterator& operator++();
102     priority_queue_iterator operator++(int);
103     priority_queue_iterator& operator--();
104     priority_queue_iterator operator--(int);
105
106     template <bool both>
107     bool operator == (priority_queue_iterator<E, R, both> const&) const;
108
109     template <bool both>
110     bool operator != (priority_queue_iterator<E, R, both> const&) const;
111
112   private:
113
114     // converters to be used by the friends
115
116     priority_queue_iterator(node_pointer, R*);
117     operator node_pointer() const;
118     operator std::string() const;
119
120     node_pointer link;
121   };
122 }
123
124 #include "priority-queue-iterator.i++"
125 #endif
```

F.6 priority-queue-iterator.i++

```
1  /*
2    Implementation of cphstl::priority_queue_iterator
3
4    Author: Jyrki Katajainen Â© 2009, 2010
5  */
6
```

```
7  #include "assert.h++"
8  #include <sstream> // defines std::stringstream
9
10 namespace cphstl {
11
12   // default constructor
13
14   template <typename E, typename R, bool is_const>
15   priority_queue_iterator<E, R, is_const>::priority_queue_iterator()
16     : link(0) {
17   }
18
19   // copy constructor
20
21   template <typename E, typename R, bool is_const>
22   priority_queue_iterator<E, R, is_const>::priority_queue_iterator(priority_queue_iterator<E,
         R, false> const& a)
23     : link(a.link) {
24   }
25
26   // assignment
27
28   template <typename E, typename R, bool is_const>
29   priority_queue_iterator<E, R, is_const>&
30   priority_queue_iterator<E, R, is_const>::operator = (priority_queue_iterator<E, R, is_const
         > const& a) {
31     link = a.link;
32     return *this;
33   }
34
35   // destructor
36
37   template <typename E, typename R, bool is_const>
38   priority_queue_iterator<E, R, is_const>::~priority_queue_iterator() {
39   }
40
41   // operator*
42
43   template <typename E, typename R, bool is_const>
44   typename priority_queue_iterator<E, R, is_const>::reference
45   priority_queue_iterator<E, R, is_const>::operator*() const {
46     return reference((*link).element());
47   }
48
49   // operator->
50
51   template <typename E, typename R, bool is_const>
52   typename priority_queue_iterator<E, R, is_const>::pointer
53   priority_queue_iterator<E, R, is_const>::operator->() const {
54     return pointer(&(*link).element());
55   }
56
57   // operator++; pre-increment
58
59   template <typename E, typename R, bool is_const>
60   priority_queue_iterator<E, R, is_const>&
61   priority_queue_iterator<E, R, is_const>::operator++() {
62     assert(link != 0);
63     node_pointer s = (*link).successor();
64     if (s != 0) {
65       link = s;
66       return *this;
67     }
68     node_pointer r = (*link).root();
69     typedef typename R::heap_store_type H;
70     typedef typename H::heap_proxy_type P;
71     P* p = (P*) (*r).owner();
72     P* q = (*p).successor();
73     if (q == 0) {
74       link = 0;
75       return *this;
76     }
77     link = (*q).root();
78     return *this;
```

```
79      }
80
81       // operator++; post-increment
82
83      template <typename E, typename R, bool is_const>
84      priority_queue_iterator<E, R, is_const>
85      priority_queue_iterator<E, R, is_const >::operator++(int) {
86        priority_queue_iterator<E, R, is_const> temporary(*this);
87        ++(*this);
88        return temporary;
89      }
90
91      // operator ==
92
93      template <typename E, typename R, bool is_const>
94      template <bool both>
95      bool
96      priority_queue_iterator<E, R, is_const >::operator == (priority_queue_iterator<E, R, both>
            const& a) const {
97        return link == a.link;
98      }
99
100     // operator !=
101
102     template <typename E, typename R, bool is_const>
103     template <bool both>
104     bool
105     priority_queue_iterator<E, R, is_const >::operator != (priority_queue_iterator<E, R, both>
            const& a) const {
106       return link != a.link;
107     }
108
109     // parametrized constructor
110
111     template <typename E, typename R, bool both>
112     priority_queue_iterator<E, R, both >::priority_queue_iterator(node_pointer p, R*)
113       : link(p) {
114     }
115
116     // conversion operators
117
118     template <typename E, typename R, bool is_const>
119     priority_queue_iterator<E, R, is_const >::operator node_pointer() const {
120       return link;
121     }
122
123     template <typename E, typename R, bool is_const>
124     priority_queue_iterator<E, R, is_const >::operator std::string() const {
125       std::stringstream ss;
126       std::string address;
127       ss << (int)(char*)((*this).link);
128       ss >> address;
129
130       if (is_const == false) {
131         return std::string("iterator: node at ") + address;
132       }
133       else {
134         return std::string("const_iterator: node at ") + address;
135       }
136     }
137
138     // pipe to an output stream
139
140     template <typename E, typename R, bool both>
141     std::ostream&
142     operator<<(std::ostream& s, priority_queue_iterator<E, R, both> const& i) {
143       s  << std::string(i);
144       return s;
145     }
146
147   }
```

F.7 proxy-iterator.h++

```
/*
  The proxy iterator stores a pointer to an encapsulator and a pointer
  to a surrogate. The pointer to the encapsulator represents the
  current position from where the value can be fetched. The surrogate
  points to a kernel. The surrogate is used to advance the iterator,
  where the access member function is called to get a pointer to the
  desired encapsulator.

  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].

  Authors: Jyrki Katajainen, Bo Simonsen Â© 2008, 2010
*/

#ifndef _CPHSTL_PROXY_ITERATOR_
#define _CPHSTL_PROXY_ITERATOR_

#include <cstddef> // std::size_t and std::ptrdiff_t
#include <iterator> // std::random_access_iterator_tag
#include "type.h++" // cphstl::if_then_else
#include <utility> // std::pair

namespace cphstl {

  template <typename V, typename A, typename K, typename I, typename J>
  class vector;

  template <typename V, typename A, typename K>
  class vector_framework;

  template <typename V, typename C, typename A, typename F, typename R,
      typename I, typename J>
  class meldable_priority_queue;

  template <typename CFG>
  class meldable_priority_queue_alt;

  template <typename E, typename R, bool is_const = false>
  class proxy_iterator {
  public:

    // types

    typedef E encapsulator_type;
    typedef R realizator_type;
    typedef std::random_access_iterator_tag iterator_category;
    typedef typename E::value_type value_type;
    typedef std::size_t size_type;
    typedef std::ptrdiff_t difference_type;
    typedef typename if_then_else<is_const, value_type const*, value_type*>::type pointer;
    typedef typename if_then_else<is_const, value_type const&, value_type&>::type reference;

  protected:

    // types

    typedef typename R::surrogate_type surrogate_type;
    typedef typename if_then_else<is_const, E const*, E*>::type node_pointer;

  public:

    // friends

    friend class proxy_iterator<E, R, !is_const>;

    template <typename V, typename A, typename K, typename I, typename J>
    friend class cphstl::vector;

    template <typename V, typename A, typename K>
    friend class cphstl::vector_framework;

    template <typename V, typename C, typename A, typename F, typename S,
        typename I, typename J>
```

```
75      friend class cphstl::meldable_priority_queue;
76
77      template <typename CFG>
78      friend class cphstl::meldable_priority_queue_alt;
79
80      // structors
81
82      proxy_iterator();
83      proxy_iterator(proxy_iterator<E, R, false> const&);
84      proxy_iterator(proxy_iterator<E, R, true> const&);
85      proxy_iterator& operator = (proxy_iterator const&);
86      ~proxy_iterator();
87
88      // operators
89
90      reference operator*() const;
91      pointer operator->() const;
92      proxy_iterator& operator++();
93      proxy_iterator operator++(int);
94      proxy_iterator& operator--();
95      proxy_iterator operator--(int);
96      proxy_iterator& operator+= (difference_type);
97      proxy_iterator& operator-= (difference_type);
98      proxy_iterator operator+(difference_type) const;
99      proxy_iterator operator-(difference_type) const;
100     difference_type operator-(proxy_iterator const&) const;
101
102     template <bool both>
103     bool operator == (proxy_iterator<E, R, both> const&) const;
104
105     template <bool both>
106     bool operator != (proxy_iterator<E, R, both> const&) const;
107
108     template <bool both>
109     bool operator<(proxy_iterator<E, R, both> const&) const;
110
111     template <bool both>
112     bool operator>(proxy_iterator<E, R, both> const&) const;
113
114     template <bool both>
115     bool operator <= (proxy_iterator<E, R, both> const&) const;
116
117     template <bool both>
118     bool operator >= (proxy_iterator<E, R, both> const&) const;
119
120   protected:
121
122     // converters to be used by the container friends
123
124     proxy_iterator(node_pointer, R*);
125     operator node_pointer() const;
126
127     template <typename S>
128     proxy_iterator(std::pair<S, surrogate_type*> const&);
129
130     template <typename S>
131     operator std::pair<S, surrogate_type*>() const;
132
133     void advance(difference_type n);
134
135     node_pointer position;
136     surrogate_type* surrogate;
137   };
138
139   template<typename E, typename R, bool both>
140   proxy_iterator<E, R, both>
141   operator+(std::ptrdiff_t, proxy_iterator<E, R, both> const&);
142
143 }
144
145 #include "proxy-iterator.i++" // implements cphstl::proxy_iterator
146
147 #endif
```

F.8 proxy-iterator.i++

```
/*
  Implementation of cphstl::proxy_iterator

  Authors: Jyrki Katajainen, Bo Simonsen Â© 2008, 2010
*/

#include "assert.h++"
#include "proxy-iterator.h++" // !!!
#include <cstdlib>
#include <iostream>

namespace cphstl {

  // default constructor

  template <typename E, typename R, bool is_const>
  proxy_iterator<E, R, is_const>::proxy_iterator()
    : position(node_pointer()), surrogate(0) {
  }

  // copy constructor

  template <typename E, typename R, bool is_const>
  proxy_iterator<E, R, is_const>::proxy_iterator(proxy_iterator<E, R, false> const& a)
    : position(a.position), surrogate(a.surrogate) {
  }

   template <typename E, typename R, bool is_const>
   proxy_iterator<E, R, is_const>::proxy_iterator(proxy_iterator<E, R, true> const& a)
     : position(a.position), surrogate(a.surrogate) {
  }

  // assignment

  template <typename E, typename R, bool is_const>
  proxy_iterator<E, R, is_const>&
  proxy_iterator<E, R, is_const>::operator = (proxy_iterator<E, R, is_const> const& a) {
    position = a.position;
    surrogate = a.surrogate;
    return *this;
  }

  // destructor

  template <typename E, typename R, bool is_const>
  proxy_iterator<E, R, is_const>::~proxy_iterator() {
  }

  // operator*

  template <typename E, typename R, bool is_const>
  typename proxy_iterator<E, R, is_const>::reference
  proxy_iterator<E, R, is_const>::operator*() const {
    return reference((*position).element());
  }

  // operator->

  template <typename E, typename R, bool is_const>
  typename proxy_iterator<E, R, is_const>::pointer
  proxy_iterator<E, R, is_const>::operator->() const {
    return pointer(&(*position).element());
  }

  template <typename E, typename R, bool is_const>
  void
  proxy_iterator<E, R, is_const>::advance(difference_type n) {
    if (n == 0) {
      return;
    }
    surrogate_type s = *surrogate;
```

```
72      if (position == 0) {
73        difference_type i = s.subject()->size() + n;
74        position = s.access(i);
75      }
76      else {
77        difference_type i = std::abs(difference_type((*position).position())) + n;
78        if (i >= difference_type(s.subject()->size()) || i < 0) {
79          position = 0;
80        }
81        else {
82          position = s.access(i);
83        }
84      }
85    }
86
87    // operator++; pre-increment
88
89    template <typename E, typename R, bool is_const>
90    proxy_iterator<E, R, is_const>&
91    proxy_iterator<E, R, is_const>::operator++() {
92      advance(1);
93      return *this;
94    }
95
96    // operator++; post-increment
97
98    template <typename E, typename R, bool is_const>
99    proxy_iterator<E, R, is_const>
100   proxy_iterator<E, R, is_const>::operator++(int) {
101     proxy_iterator<E, R, is_const> temporary = *this;
102     advance(1);
103     return temporary;
104   }
105
106   // operator--; pre-decrement
107
108   template <typename E, typename R, bool is_const>
109   proxy_iterator<E, R, is_const>&
110   proxy_iterator<E, R, is_const>::operator--() {
111     advance(-1);
112     return *this;
113   }
114
115   // operator--; post-decrement
116
117   template <typename E, typename R, bool is_const>
118   proxy_iterator<E, R, is_const>
119   proxy_iterator<E, R, is_const>::operator--(int) {
120     proxy_iterator<E, R, is_const> temporary = *this;
121     advance(-1);
122     return temporary;
123   }
124
125   // operator+=
126
127   template <typename E, typename R, bool is_const>
128   proxy_iterator<E, R, is_const>&
129   proxy_iterator<E, R, is_const>::operator+= (difference_type n) {
130     advance(n);
131     return *this;
132   }
133
134   // operator-=
135
136   template <typename E, typename R, bool is_const>
137   proxy_iterator<E, R, is_const>&
138   proxy_iterator<E, R, is_const>::operator-= (difference_type n) {
139     advance(-n);
140     return *this;
141   }
142
143   // operator+
144
145   template <typename E, typename R, bool is_const>
```

```
146     proxy_iterator<E, R, is_const>
147     proxy_iterator<E, R, is_const>::operator+(difference_type n) const {
148       proxy_iterator<E, R, is_const> temporary = *this;
149       temporary.advance(n);
150       return temporary;
151     }
152
153     // operator-
154
155     template <typename E, typename R, bool is_const>
156     proxy_iterator<E, R, is_const>
157     proxy_iterator<E, R, is_const>::operator-(difference_type n) const {
158       proxy_iterator<E, R, is_const> temporary = *this;
159       temporary.advance(-n);
160       return temporary;
161     }
162
163     // iterator distance
164
165     template <typename E, typename R, bool is_const>
166     typename proxy_iterator<E, R, is_const>::difference_type
167     proxy_iterator<E, R, is_const>::operator-(proxy_iterator<E, R, is_const> const& a) const {
168       if (position == 0 && a.position == 0) {
169         return 0;
170       }
171       else if (position == 0) {
172         difference_type n = (*(*surrogate).subject()).size();
173         difference_type i = std::abs(difference_type((*a.position).position()));
174         return n - i;
175       }
176       else if (a.position == 0) {
177         difference_type i = std::abs(difference_type((*position).position()));
178         difference_type n = (*(*surrogate).subject()).size();
179         return i - n;
180       }
181       else {
182         difference_type i = std::abs(difference_type((*position).position()));
183         difference_type j = std::abs(difference_type((*a.position).position()));
184         return i - j;
185       }
186     }
187
188     // operator ==
189
190     template <typename E, typename R, bool is_const>
191     template <bool both>
192     bool
193     proxy_iterator<E, R, is_const>::operator == (proxy_iterator<E, R, both> const& a) const {
194       return position == a.position && surrogate == a.surrogate;
195     }
196
197     // operator !=
198
199     template <typename E, typename R, bool is_const>
200     template <bool both>
201     bool
202     proxy_iterator<E, R, is_const>::operator != (proxy_iterator<E, R, both> const& a) const {
203       return !(*this == a);
204     }
205
206     // operator<
207
208     template <typename E, typename R, bool is_const>
209     template <bool both>
210     bool
211     proxy_iterator<E, R, is_const>::operator<(proxy_iterator<E, R, both> const& a) const {
212       return ((*this) - a) < 0;
213     }
214
215     // operator>
216
217     template <typename E, typename R, bool is_const>
218     template <bool both>
219     bool
```

```
220     proxy_iterator<E, R, is_const>::operator>(proxy_iterator<E, R, both> const& a) const {
221       return a < *this;
222     }
223
224     // operator<=
225
226     template <typename E, typename R, bool is_const>
227     template <bool both>
228     bool
229     proxy_iterator<E, R, is_const>::operator<= (proxy_iterator<E, R, both> const& a) const {
230       return !(a < *this);
231     }
232
233     // operator>=
234
235     template <typename E, typename R, bool is_const>
236     template <bool both>
237     bool
238     proxy_iterator<E, R, is_const>::operator>= (proxy_iterator<E, R, both> const& a) const {
239       return !(*this < a);
240     }
241
242     // parametrized constructors
243
244     template <typename E, typename R, bool is_const>
245     proxy_iterator<E, R, is_const>::proxy_iterator(node_pointer p, realizator_type* r)
246       : position(p), surrogate((*r).surrogate) {
247     }
248
249     template <typename E, typename R, bool is_const>
250     template <typename S>
251     proxy_iterator<E, R, is_const>::proxy_iterator(std::pair<S, surrogate_type*> const& p)
252       : position(0), surrogate(p.second) {
253       // convert index to a proxy
254     }
255
256     // convertion operators
257
258     template <typename E, typename R, bool is_const>
259     proxy_iterator<E, R, is_const>::operator node_pointer() const {
260       return position;
261     }
262
263     template <typename E, typename R, bool is_const>
264     template <typename S>
265     proxy_iterator<E, R, is_const>::operator std::pair<S, /*!!!*/ typename proxy_iterator<E, R
            , is_const>::surrogate_type*>() const {
266       // convert proxy to an index
267       return std::pair<S, surrogate_type*>(S(0), (*this).surrogate);
268     }
269
270     // operator+(int, iterator)
271
272     template <typename E, typename R, bool is_const>
273     proxy_iterator<E, R, is_const> operator+(std::ptrdiff_t n, proxy_iterator<E, R, is_const>
            const& a) {
274       return a + n;
275     }
276
277 }
```

F.9 rank-iterator.h++

```
1  /*
2    A rank-based iterator is just a (pointer, index) pair where the
3    pointer points to the data structure containing the cell referred to
4    and the index is the rank of that cell in the sequence of cells
5    storing the elements.
6
7    The idea of combining iterators and const iterators into the same
8    class is taken from [Matt Austern. Defining iterators and const
9    iterators. C/C++ User's Journal 19,1 (2001), 74-79].
10
```

```
11    Authors: Jyrki Katajainen, Bo Simonsen Â© 2008
12  */
13
14  #ifndef _CPHSTL_RANK_ITERATOR_
15  #define _CPHSTL_RANK_ITERATOR_
16
17  #include <cstddef> // defines std::size_t and std::ptrdiff_t
18  #include <iterator> // defines std::random_access_iterator_tag
19  #include "type.h++" // defines cphstl::if_then_else
20  #include <utility> // defines std::pair
21
22  namespace cphstl {
23
24    template <typename V, typename A, typename R, typename I, typename J>
25    class vector;
26
27    template <typename R, bool is_const = false, typename E = typename R::encapsulator_type
          >
28    class rank_iterator {
29
30    public:
31
32      // types
33
34      typedef std::random_access_iterator_tag iterator_category;
35      typedef typename R::value_type value_type;
36      typedef std::size_t size_type;
37      typedef std::ptrdiff_t difference_type;
38      typedef typename if_then_else<is_const, value_type const*, value_type*>::type pointer;
39      typedef typename if_then_else<is_const, typename R::const_reference, typename R::
            reference>::type reference;
40
41      typedef E entry;
42      typedef typename R::surrogate_type surrogate_type;
43
44    protected:
45
46      // types
47
48      typedef typename if_then_else<is_const, E const*, E*>::type node_pointer;
49
50    public:
51
52      // friends
53
54      friend class rank_iterator<R, !is_const, E>;
55
56      template <typename V, typename A, typename S, typename I, typename J>
57      friend class cphstl::vector;
58
59      // structors
60
61      rank_iterator();
62      rank_iterator(rank_iterator<R, false, E> const&);
63      rank_iterator(rank_iterator<R, true, E> const&);
64      rank_iterator& operator = (rank_iterator const&);
65      ~rank_iterator();
66
67      // operators
68
69      reference operator*() const;
70      pointer operator->() const;
71      rank_iterator& operator++();
72      rank_iterator operator++(int);
73      rank_iterator& operator--();
74      rank_iterator operator--(int);
75      rank_iterator& operator+= (difference_type);
76      rank_iterator& operator-= (difference_type);
77      rank_iterator operator+(difference_type) const;
78      rank_iterator operator-(difference_type) const;
79      difference_type operator-(rank_iterator const&) const;
80
81      template <bool both>
82      bool operator == (rank_iterator<R, both, E> const&) const;
```

```
83
84      template <bool both>
85      bool operator != (rank_iterator<R, both, E> const&) const;
86
87      template <bool both>
88      bool operator<(rank_iterator<R, both, E> const&) const;
89
90      template <bool both>
91      bool operator>(rank_iterator<R, both, E> const&) const;
92
93      template <bool both>
94      bool operator<= (rank_iterator<R, both, E> const&) const;
95
96      template <bool both>
97      bool operator>= (rank_iterator<R, both, E> const&) const;
98
99    protected:
100     // converters to be used by the container friends
101     rank_iterator(std::pair<size_type, surrogate_type*> const&);
102     operator std::pair<size_type, surrogate_type*>() const;
103
104     void advance(difference_type const& n);
105
106     surrogate_type* surrogate;
107     size_type position;
108   };
109
110   template<typename R, bool both, typename E>
111   rank_iterator<R, both, E>
112   operator+(typename R::difference_type, rank_iterator<R, both, E> const&);
113
114 }
115
116 #include "rank-iterator.i++" // implements cphstl::rank_iterator
117
118 #endif
```

F.10 rank-iterator.i++

```
1  /*
2    Implementation of cphstl::rank_iterator
3
4    Authors: Jyrki Katajainen, Bo Simonsen Â© 2008
5  */
6
7  #include "assert.h++"
8  #include <iostream>
9
10 namespace cphstl {
11
12   // default constructor
13
14   template <typename R, bool is_const, typename E>
15   rank_iterator<R, is_const, E>::rank_iterator()
16     : surrogate(0), position(size_type()) {
17   }
18
19   // copy constructor
20
21   template <typename R, bool is_const, typename E>
22   rank_iterator<R, is_const, E>::rank_iterator(rank_iterator<R, false, E> const& a)
23     : surrogate(a.surrogate), position(a.position) {
24   }
25
26   template <typename R, bool is_const, typename E>
27   rank_iterator<R, is_const, E>::rank_iterator(rank_iterator<R, true, E> const& a)
28     : surrogate(a.surrogate), position(a.position) {
29   }
30
31   // assignment
32
33   template <typename R, bool is_const, typename E>
34   rank_iterator<R, is_const, E>&
```

```
35    rank_iterator<R, is_const, E>::operator = (rank_iterator<R, is_const, E> const& a) {
36      (*this).surrogate = a.surrogate;
37      (*this).position = a.position;
38      return *this;
39    }
40
41    // destructor
42
43    template <typename R, bool is_const, typename E>
44    rank_iterator<R, is_const, E>::~rank_iterator() {
45    }
46
47    // operator*
48
49    template <typename R, bool is_const, typename E>
50    typename rank_iterator<R, is_const, E>::reference
51    rank_iterator<R, is_const, E>::operator*() const {
52      return reference((*(*(*this).surrogate).subject()).access((*this).position));
53    }
54
55    // operator->
56
57    template <typename R, bool is_const, typename E>
58    typename rank_iterator<R, is_const, E>::pointer
59    rank_iterator<R, is_const, E>::operator->() const {
60      return pointer(&(*(*this).position).content());
61    }
62
63    template <typename R, bool is_const, typename E>
64    void
65    rank_iterator<R, is_const, E>::advance(difference_type const& n) {
66      if (n == 0)
67        return;
68      (*this).position += n;
69    }
70
71    // operator++; pre-increment
72
73    template <typename R, bool is_const, typename E>
74    rank_iterator<R, is_const, E>&
75    rank_iterator<R, is_const, E>::operator++() {
76      (*this).advance(1);
77      return *this;
78    }
79
80    // operator++; post-increment
81
82    template <typename R, bool is_const, typename E>
83    rank_iterator<R, is_const, E>
84    rank_iterator<R, is_const, E>::operator++(int) {
85      rank_iterator<R, is_const, E> temporary = *this;
86      (*this).advance(1);
87      return temporary;
88    }
89
90    // operator--; pre-decrement
91
92    template <typename R, bool is_const, typename E>
93    rank_iterator<R, is_const, E>&
94    rank_iterator<R, is_const, E>::operator--() {
95      (*this).advance(-1);
96      return *this;
97    }
98
99    // operator--; post-decrement
100
101   template <typename R, bool is_const, typename E>
102   rank_iterator<R, is_const, E>
103   rank_iterator<R, is_const, E>::operator--(int) {
104     rank_iterator<R, is_const, E> temporary = *this;
105     (*this).advance(-1);
106     return temporary;
107   }
108
```

```
109     // operator+=
110
111     template <typename R, bool is_const, typename E>
112     rank_iterator<R, is_const, E>&
113     rank_iterator<R, is_const, E>::operator+= (difference_type n) {
114       (*this).advance(n);
115       return *this;
116     }
117
118     // operator-=
119
120     template <typename R, bool is_const, typename E>
121     rank_iterator<R, is_const, E>&
122     rank_iterator<R, is_const, E>::operator-= (difference_type n) {
123       (*this).advance(-n);
124       return *this;
125     }
126
127     // operator+
128
129     template <typename R, bool is_const, typename E>
130     rank_iterator<R, is_const, E>
131     rank_iterator<R, is_const, E>::operator+(difference_type n) const {
132       rank_iterator<R, is_const, E> temporary = *this;
133       temporary.advance(n);
134       return temporary;
135     }
136
137     // operator-
138
139     template <typename R, bool is_const, typename E>
140     rank_iterator<R, is_const, E>
141     rank_iterator<R, is_const, E>::operator-(difference_type n) const {
142       rank_iterator<R, is_const, E> temporary = *this;
143       temporary.advance(-n);
144       return temporary;
145     }
146
147     // iterator distance
148
149     template <typename R, bool is_const, typename E>
150     typename rank_iterator<R, is_const, E>::difference_type
151     rank_iterator<R, is_const, E>::operator-(rank_iterator<R, is_const, E> const& a) const {
152       return (*this).position - a.position;
153     }
154
155     // operator ==
156
157     template <typename R, bool is_const, typename E>
158     template <bool both>
159     bool
160     rank_iterator<R, is_const, E>::operator == (rank_iterator<R, both, E> const& a) const {
161       return (*this).position == a.position;
162     }
163
164     // operator !=
165
166     template <typename R, bool is_const, typename E>
167     template <bool both>
168     bool
169     rank_iterator<R, is_const, E>::operator != (rank_iterator<R, both, E> const& a) const {
170       return !(*this == a);
171     }
172
173     // operator<
174
175     template <typename R, bool is_const, typename E>
176     template <bool both>
177     bool
178     rank_iterator<R, is_const, E>::operator<(rank_iterator<R, both, E> const& a) const {
179       return ((*this) - a) < 0;
180     }
181
182     // operator>
```

```
183
184     template <typename R, bool is_const, typename E>
185     template <bool both>
186     bool
187     rank_iterator<R, is_const, E>::operator>(rank_iterator<R, both, E> const& a) const {
188       return a < *this;
189     }
190
191     // operator<=
192
193     template <typename R, bool is_const, typename E>
194     template <bool both>
195     bool
196     rank_iterator<R, is_const, E>::operator<= (rank_iterator<R, both, E> const& a) const {
197       return !(a < *this);
198     }
199
200     // operator>=
201
202     template <typename R, bool is_const, typename E>
203     template <bool both>
204     bool
205     rank_iterator<R, is_const, E>::operator>= (rank_iterator<R, both, E> const& a) const {
206       return !(*this < a);
207     }
208
209     // operator+(int, iterator)
210
211     template <typename R, bool is_const, typename E>
212     rank_iterator<R, is_const, E> operator+(typename R::difference_type n, rank_iterator<R,
          is_const, E> const& a) {
213       return a + n;
214     }
215
216     // parametrized constructor
217
218     template <typename R, bool is_const, typename E>
219     rank_iterator<R, is_const, E>::rank_iterator(std::pair<size_type, surrogate_type*> const& p)
220       : surrogate(p.second), position(p.first) {
221
222     }
223
224     // convertion operator
225
226     template <typename R, bool is_const, typename E>
227     rank_iterator<R, is_const, E>::operator std::pair<size_type, surrogate_type*>() const {
228       return std::pair<size_type, surrogate_type*>((*this).position, (*this).surrogate);
229     }
230   }
```

F.11 unidirectional-composite-iterator.h++

```
1   /*
2     The proxy iterator stores a pointer to an encapsulator and a pointer
3     to a surrogate. The pointer to the encapsulator represents the
4     current position from where the value can be fetched. The surrogate
5     points to a kernel. The surrogate is used to advance the iterator,
6     where the access member function is called to get a pointer to the
7     desired encapsulator.
8
9     The idea of combining iterators and const iterators into the same
10    class is taken from [Matt Austern. Defining iterators and const
11    iterators. C/C++ User's Journal 19,1 (2001), 74-79].
12
13    Authors: Jyrki Katajainen, Bo Simonsen Â© 2008, 2010
14  */
15
16  #ifndef _CPHSTL_PROXY_ITERATOR_
17  #define _CPHSTL_PROXY_ITERATOR_
18
19  #include <cstddef> // std::size_t and std::ptrdiff_t
20  #include <iterator> // std::random_access_iterator_tag
21  #include "type.h++" // cphstl::if_then_else
```

```
22 #include <utility> // std::pair
23
24 namespace cphstl {
25
26   template <typename V, typename A, typename K, typename I, typename J>
27   class vector;
28
29   template <typename V, typename A, typename K>
30   class vector_framework;
31
32   template <typename V, typename C, typename A, typename F, typename R,
33       typename I, typename J>
34   class meldable_priority_queue;
35
36   template <typename E, typename R, bool is_const = false>
37   class proxy_iterator {
38   public:
39
40     // types
41
42     typedef E encapsulator_type;
43     typedef R realizator_type;
44     typedef std::random_access_iterator_tag iterator_category;
45     typedef typename E::value_type value_type;
46     typedef std::size_t size_type;
47     typedef std::ptrdiff_t difference_type;
48     typedef typename if_then_else<is_const, value_type const*, value_type*>::type pointer;
49     typedef typename if_then_else<is_const, value_type const&, value_type&>::type reference;
50
51   protected:
52
53     // types
54
55     typedef typename R::surrogate_type surrogate_type;
56     typedef typename if_then_else<is_const, E const*, E*>::type node_pointer;
57
58   public:
59
60     // friends
61
62     friend class proxy_iterator<E, R, !is_const>;
63
64     template <typename V, typename A, typename K, typename I, typename J>
65     friend class cphstl::vector;
66
67     template <typename V, typename A, typename K>
68     friend class cphstl::vector_framework;
69
70     template <typename V, typename C, typename A, typename F, typename S,
71         typename I, typename J>
72     friend class cphstl::meldable_priority_queue;
73
74     // structors
75
76     proxy_iterator();
77     proxy_iterator(proxy_iterator<E, R, false> const&);
78     proxy_iterator(proxy_iterator<E, R, true> const&);
79     proxy_iterator& operator = (proxy_iterator const&);
80     ~proxy_iterator();
81
82     // operators
83
84     reference operator*() const;
85     pointer operator->() const;
86     proxy_iterator& operator++();
87     proxy_iterator operator++(int);
88     proxy_iterator& operator--();
89     proxy_iterator operator--(int);
90     proxy_iterator& operator+= (difference_type);
91     proxy_iterator& operator-= (difference_type);
92     proxy_iterator operator+(difference_type) const;
93     proxy_iterator operator-(difference_type) const;
94     difference_type operator-(proxy_iterator const&) const;
95
```

```
96      template <bool both>
97      bool operator == (proxy_iterator<E, R, both> const&) const;
98
99      template <bool both>
100     bool operator != (proxy_iterator<E, R, both> const&) const;
101
102     template <bool both>
103     bool operator<(proxy_iterator<E, R, both> const&) const;
104
105     template <bool both>
106     bool operator>(proxy_iterator<E, R, both> const&) const;
107
108     template <bool both>
109     bool operator<= (proxy_iterator<E, R, both> const&) const;
110
111     template <bool both>
112     bool operator>= (proxy_iterator<E, R, both> const&) const;
113
114   protected:
115
116     // converters to be used by the container friends
117
118     proxy_iterator(node_pointer, R*);
119     operator node_pointer() const;
120
121     template <typename S>
122     proxy_iterator(std::pair<S, surrogate_type*> const&);
123
124     template <typename S>
125     operator std::pair<S, surrogate_type*>() const;
126
127     void advance(difference_type n);
128
129     node_pointer position;
130     surrogate_type* surrogate;
131   };
132
133   template<typename E, typename R, bool both>
134   proxy_iterator<E, R, both>
135   operator+(std::ptrdiff_t, proxy_iterator<E, R, both> const&);
136
137 }
138
139 #include "proxy-iterator.i++" // implements cphstl::proxy_iterator
140
141 #endif
```

F.12 unidirectional-composite-iterator.i++

```
1  /*
2    Implementation of cphstl::proxy_iterator
3
4    Authors: Jyrki Katajainen, Bo Simonsen Â© 2008, 2010
5  */
6
7  #include "assert.h++"
8  #include <cstdlib>
9  #include <iostream>
10
11 namespace cphstl {
12
13   // default constructor
14
15   template <typename E, typename R, bool is_const>
16   proxy_iterator<E, R, is_const>::proxy_iterator()
17     : position(node_pointer()), surrogate(0) {
18   }
19
20   // copy constructor
21
22   template <typename E, typename R, bool is_const>
23   proxy_iterator<E, R, is_const>::proxy_iterator(proxy_iterator<E, R, false> const& a)
24     : position(a.position), surrogate(a.surrogate) {
```

```
25      }
26
27       template <typename E, typename R, bool is_const>
28       proxy_iterator<E, R, is_const>::proxy_iterator(proxy_iterator<E, R, true> const& a)
29         : position(a.position), surrogate(a.surrogate) {
30      }
31
32      // assignment
33
34      template <typename E, typename R, bool is_const>
35      proxy_iterator<E, R, is_const>&
36      proxy_iterator<E, R, is_const>::operator = (proxy_iterator<E, R, is_const> const& a) {
37        position = a.position;
38        surrogate = a.surrogate;
39        return *this;
40      }
41
42      // destructor
43
44      template <typename E, typename R, bool is_const>
45      proxy_iterator<E, R, is_const>::~proxy_iterator() {
46      }
47
48      // operator*
49
50      template <typename E, typename R, bool is_const>
51      typename proxy_iterator<E, R, is_const>::reference
52      proxy_iterator<E, R, is_const>::operator*() const {
53        return reference((*position).element());
54      }
55
56      // operator->
57
58      template <typename E, typename R, bool is_const>
59      typename proxy_iterator<E, R, is_const>::pointer
60      proxy_iterator<E, R, is_const>::operator->() const {
61        return pointer(&(*position).element());
62      }
63
64      template <typename E, typename R, bool is_const>
65      void
66      proxy_iterator<E, R, is_const>::advance(difference_type n) {
67        if (n == 0) {
68          return;
69        }
70        surrogate_type s = *surrogate;
71        if (position == 0) {
72          difference_type i = s.subject()->size() + n;
73          position = s.access(i);
74        }
75        else {
76          difference_type i = std::abs(difference_type((*position).position())) + n;
77          if (i >= difference_type(s.subject()->size()) || i < 0) {
78            position = 0;
79          }
80          else {
81            position = s.access(i);
82          }
83        }
84      }
85
86      // operator++; pre-increment
87
88      template <typename E, typename R, bool is_const>
89      proxy_iterator<E, R, is_const>&
90      proxy_iterator<E, R, is_const>::operator++() {
91        advance(1);
92        return *this;
93      }
94
95      // operator++; post-increment
96
97      template <typename E, typename R, bool is_const>
98      proxy_iterator<E, R, is_const>
```

```
99    proxy_iterator<E, R, is_const>::operator++(int) {
100     proxy_iterator<E, R, is_const> temporary = *this;
101     advance(1);
102     return temporary;
103   }
104
105   // operator--; pre-decrement
106
107   template <typename E, typename R, bool is_const>
108   proxy_iterator<E, R, is_const>&
109   proxy_iterator<E, R, is_const>::operator--() {
110     advance(-1);
111     return *this;
112   }
113
114   // operator--; post-decrement
115
116   template <typename E, typename R, bool is_const>
117   proxy_iterator<E, R, is_const>
118   proxy_iterator<E, R, is_const>::operator--(int) {
119     proxy_iterator<E, R, is_const> temporary = *this;
120     advance(-1);
121     return temporary;
122   }
123
124   // operator+=
125
126   template <typename E, typename R, bool is_const>
127   proxy_iterator<E, R, is_const>&
128   proxy_iterator<E, R, is_const>::operator+= (difference_type n) {
129     advance(n);
130     return *this;
131   }
132
133   // operator-=
134
135   template <typename E, typename R, bool is_const>
136   proxy_iterator<E, R, is_const>&
137   proxy_iterator<E, R, is_const>::operator-= (difference_type n) {
138     advance(-n);
139     return *this;
140   }
141
142   // operator+
143
144   template <typename E, typename R, bool is_const>
145   proxy_iterator<E, R, is_const>
146   proxy_iterator<E, R, is_const>::operator+(difference_type n) const {
147     proxy_iterator<E, R, is_const> temporary = *this;
148     temporary.advance(n);
149     return temporary;
150   }
151
152   // operator-
153
154   template <typename E, typename R, bool is_const>
155   proxy_iterator<E, R, is_const>
156   proxy_iterator<E, R, is_const>::operator-(difference_type n) const {
157     proxy_iterator<E, R, is_const> temporary = *this;
158     temporary.advance(-n);
159     return temporary;
160   }
161
162   // iterator distance
163
164   template <typename E, typename R, bool is_const>
165   typename proxy_iterator<E, R, is_const>::difference_type
166   proxy_iterator<E, R, is_const>::operator-(proxy_iterator<E, R, is_const> const& a) const {
167     if (position == 0 && a.position == 0) {
168       return 0;
169     }
170     else if (position == 0) {
171       difference_type n = (*(*surrogate).subject()).size();
172       difference_type i = std::abs(difference_type((*a.position).position()));
```

```
173         return n - i;
174       }
175       else if (a.position == 0) {
176         difference_type i = std::abs(difference_type((*position).position()));
177         difference_type n = (*(*surrogate).subject()).size();
178         return i - n;
179       }
180       else {
181         difference_type i = std::abs(difference_type((*position).position()));
182         difference_type j = std::abs(difference_type((*a.position).position()));
183         return i - j;
184       }
185     }
186
187     // operator ==
188
189     template <typename E, typename R, bool is_const>
190     template <bool both>
191     bool
192     proxy_iterator<E, R, is_const>::operator == (proxy_iterator<E, R, both> const& a) const {
193       return position == a.position && surrogate == a.surrogate;
194     }
195
196     // operator !=
197
198     template <typename E, typename R, bool is_const>
199     template <bool both>
200     bool
201     proxy_iterator<E, R, is_const>::operator != (proxy_iterator<E, R, both> const& a) const {
202       return !(*this == a);
203     }
204
205     // operator<
206
207     template <typename E, typename R, bool is_const>
208     template <bool both>
209     bool
210     proxy_iterator<E, R, is_const>::operator<(proxy_iterator<E, R, both> const& a) const {
211       return ((*this) - a) < 0;
212     }
213
214     // operator>
215
216     template <typename E, typename R, bool is_const>
217     template <bool both>
218     bool
219     proxy_iterator<E, R, is_const>::operator>(proxy_iterator<E, R, both> const& a) const {
220       return a < *this;
221     }
222
223     // operator<=
224
225     template <typename E, typename R, bool is_const>
226     template <bool both>
227     bool
228     proxy_iterator<E, R, is_const>::operator<= (proxy_iterator<E, R, both> const& a) const {
229       return !(a < *this);
230     }
231
232     // operator>=
233
234     template <typename E, typename R, bool is_const>
235     template <bool both>
236     bool
237     proxy_iterator<E, R, is_const>::operator>= (proxy_iterator<E, R, both> const& a) const {
238       return !(*this < a);
239     }
240
241     // parametrized constructors
242
243     template <typename E, typename R, bool is_const>
244     proxy_iterator<E, R, is_const>::proxy_iterator(node_pointer p, realizator_type* r)
245       : position(p), surrogate((*r).surrogate) {
246     }
```

```
247
248   template <typename E, typename R, bool is_const>
249   template <typename S>
250   proxy_iterator<E, R, is_const>::proxy_iterator(std::pair<S, surrogate_type*> const& p)
251     : position(0), surrogate(p.second) {
252     // convert index to a proxy
253   }
254
255   // convertion operators
256
257   template <typename E, typename R, bool is_const>
258   proxy_iterator<E, R, is_const>::operator node_pointer() const {
259     return position;
260   }
261
262   template <typename E, typename R, bool is_const>
263   template <typename S>
264   proxy_iterator<E, R, is_const>::operator std::pair<S, surrogate_type*>() const {
265     // convert proxy to an index
266     return std::pair<S, surrogate_type*>(S(0), (*this).surrogate);
267   }
268
269   // operator+(int, iterator)
270
271   template <typename E, typename R, bool is_const>
272   proxy_iterator<E, R, is_const> operator+(std::ptrdiff_t n, proxy_iterator<E, R, is_const>
          const& a) {
273     return a + n;
274   }
275
276 }
```

F.13 unidirectional-monolith-iterator.h++

```
1  /*
2    The idea of combining iterators and const iterators into the same
3    class is taken from [Matt Austern. Defining iterators and const
4    iterators. C/C++ User's Journal 19,1 (2001), 74-79].
5
6    Authors: Jyrki Katajainen, Bo Simonsen, 2006, 2008
7  */
8
9  #ifndef _CPHSTL_NODE_ITERATOR_
10 #define _CPHSTL_NODE_ITERATOR_
11
12 #include <cstddef> // std::ptrdiff_t
13 #include <iostream> // std::ostream
14 #include <iterator> // std::bidirectional_iterator_tag
15 #include <string> // std::string
16 #include "type.h++" // cphstl::if_then_else
17
18 namespace leda {
19   template < typename K, typename I, typename C, typename R >
20   class dictionary;
21 }
22
23 namespace cphstl {
24
25   /* Forward declarations of container classes */
26
27   template <typename V, typename A, typename R, typename I, typename J>
28   class list;
29
30   template <typename V, typename C, typename A, typename R,
31       typename I, typename J>
32   class multiset;
33
34   template <typename V, typename C, typename A, typename R,
35       typename I, typename J>
36   class set;
37
38   template <typename K, typename V, typename C, typename A, typename R,
39       typename I, typename J>
```

```
40      class set_base;
41
42      template <typename K, typename V, typename C, typename A, typename R,
43          typename I, typename J>
44      class map;
45
46      template <typename K, typename V, typename C, typename A, typename R,
47          typename I, typename J>
48      class map_base;
49
50
51      template <typename V, typename C, typename A, typename E,
52        typename R, typename I, typename J>
53      class meldable_priority_queue;
54
55
56      template <typename N, bool is_const = false>
57      class node_iterator {
58
59      public:
60
61        // types
62
63        typedef std::bidirectional_iterator_tag iterator_category;
64        typedef typename N::value_type value_type;
65        typedef std::ptrdiff_t difference_type;
66
67        typedef typename if_then_else<is_const, value_type const*, value_type*>::type pointer;
68        typedef typename if_then_else<is_const, value_type const&, value_type&>::type reference;
69
70        typedef node_iterator<N, !is_const> complement;
71      private:
72
73        // types
74
75        typedef typename if_then_else<is_const, N const*, N*>::type node_pointer;
76
77      public:
78
79        // friends
80        friend class node_iterator<N, !is_const>;
81
82        template <typename M, bool both>
83        friend std::ostream& operator<<(std::ostream&, node_iterator<M, both> const&);
84
85        template <typename V, typename A, typename R, typename I, typename J>
86        friend class cphstl::list;
87
88        template <typename V, typename C, typename A, typename R,
89            typename I, typename J>
90        friend class cphstl::set;
91
92        template <typename K, typename V, typename C, typename A, typename R,
93            typename I, typename J>
94        friend class cphstl::set_base;
95
96        template <typename V, typename C, typename A, typename R,
97            typename I, typename J>
98        friend class cphstl::multiset;
99
100       template <typename K, typename V, typename C, typename A,
101           typename R, typename I, typename J>
102       friend class cphstl::map;
103
104       template <typename K, typename V, typename C, typename A, typename R,
105           typename I, typename J>
106       friend class cphstl::map_base;
107
108
109       template <typename V, typename C, typename A, typename E,
110         typename R, typename I, typename J>
111         friend class cphstl::meldable_priority_queue;
112
113
```

```
114       template <typename T1, typename T2>
115       friend class std::pair;
116
117       template < typename K, typename I, typename C, typename R >
118       friend class leda::dictionary;
119
120       // structors
121
122       node_iterator();
123       node_iterator(node_iterator<N, false> const&);
124       template<typename R>
125       node_iterator(node_iterator<N, false> const&, R const&);
126       node_iterator& operator = (node_iterator const&);
127       ~node_iterator();
128
129       // operators
130
131       reference operator*() const;
132       pointer operator->() const;
133       pointer operator->();
134       node_iterator& operator++();
135       node_iterator operator++(int);
136       node_iterator& operator--();
137       node_iterator operator--(int);
138
139       template <bool both>
140       bool operator == (node_iterator<N, both> const&) const;
141
142       template <bool both>
143       bool operator != (node_iterator<N, both> const&) const;
144
145     private:
146       // converters to be used by the friends
147
148       node_iterator(node_pointer); // node_pointer --> iterator
149       operator node_pointer() const; // iterator --> node_pointer
150       operator std::string() const; // iterator --> string
151
152       node_pointer link;
153
154     };
155
156   }
157
158   #include "node-iterator.i++" // implements cphstl::node_iterator
159
160   #endif
```

F.14 unidirectional-monolith-iterator.i++

```
1   /*
2     Implementation of cphstl::node_iterator
3
4     Author: Jyrki Katajainen, Bo Simonsen, April 2008
5
6     General design idea is proposed by J. Katajainen and B. Simonsen
7   */
8
9   #include <sstream> // defines std::stringstream
10
11  namespace cphstl {
12
13    // default constructor
14
15    template <typename N, bool is_const>
16    node_iterator<N, is_const>::node_iterator()
17      : link(0) {
18    }
19
20    // copy constructor
21
22    template <typename N, bool is_const>
23    node_iterator<N, is_const>::node_iterator(node_iterator<N, false> const& a)
```

```
24      : link(a.link) {
25    }
26    template <typename N, bool is_const>
27    template<typename R>
28    node_iterator<N, is_const>::node_iterator(node_iterator<N, false> const& a, R const&)
29      : link(a.link) {
30    }
31
32
33    // assignment
34
35    template <typename N, bool is_const>
36    node_iterator<N, is_const>&
37    node_iterator<N, is_const>::operator = (node_iterator<N, is_const> const& a) {
38      link = a.link;
39      return *this;
40    }
41
42    // destructor
43
44    template <typename N, bool is_const>
45    node_iterator<N, is_const>::~node_iterator() {
46    }
47
48    // operator*
49
50    template <typename N, bool is_const>
51    typename node_iterator<N, is_const>::reference
52    node_iterator<N, is_const>::operator*() const {
53      return reference((*link).content());
54    }
55
56    // operator->
57
58    template <typename N, bool is_const>
59    typename node_iterator<N, is_const>::pointer
60    node_iterator<N, is_const>::operator->() const {
61      return pointer(&(*link).content());
62    }
63
64    template <typename N, bool is_const>
65    typename node_iterator<N, is_const>::pointer
66    node_iterator<N, is_const>::operator->() {
67      return pointer(&(*link).content());
68    }
69
70    // operator++; pre-increment
71
72    template <typename N, bool is_const>
73    node_iterator<N, is_const>&
74    node_iterator<N, is_const>::operator++() {
75      link = (*link).successor();
76      return *this;
77    }
78
79     // operator++; post-increment
80
81    template <typename N, bool is_const>
82    node_iterator<N, is_const>
83    node_iterator<N, is_const>::operator++(int) {
84      node_iterator<N, is_const> temporary(*this);
85      ++(*this);
86      return temporary;
87    }
88
89    // operator--; pre-increment
90
91    template <typename N, bool is_const>
92    node_iterator<N, is_const>&
93    node_iterator<N, is_const>::operator--() {
94      link = (*link).predecessor();
95      return *this;
96    }
97
```

```
98      // operator--; post-increment
99
100     template <typename N, bool is_const>
101     node_iterator<N, is_const>
102     node_iterator<N, is_const>::operator--(int) {
103       node_iterator<N, is_const> temporary(*this);
104       --(*this);
105       return temporary;
106     }
107
108     // operator ==
109
110     template <typename N, bool is_const>
111     template <bool both>
112     bool
113     node_iterator<N, is_const>::operator == (node_iterator<N, both> const& a) const {
114       return link == a.link;
115     }
116
117     // operator !=
118
119     template <typename N, bool is_const>
120     template <bool both>
121     bool
122     node_iterator<N, is_const>::operator != (node_iterator<N, both> const& a) const {
123       return link != a.link;
124     }
125
126     // parametrized constructor (node -> iterator)
127
128     template <typename N, bool both>
129     node_iterator<N, both>::node_iterator(node_pointer p)
130       : link(p) {
131     }
132
133     // conversion operators (iterator -> node)
134
135     template <typename N, bool is_const>
136     node_iterator<N, is_const>::operator node_pointer() const {
137       return link;
138     }
139
140     // conversion operator (makes it possible to print out an iterator)
141
142     template <typename N, bool is_const>
143     node_iterator<N, is_const>::operator std::string() const {
144       std::stringstream ss;
145       std::string address;
146       ss << (int)(char*)((*this).link);
147       ss >> address;
148
149       if (is_const == false) {
150         return std::string("iterator: node at ") + address;
151       }
152       else {
153         return std::string("const_iterator: node at ") + address;
154       }
155     }
156
157     // representation
158
159     template <typename N, bool both>
160     std::ostream&
161     operator<<(std::ostream& s, node_iterator<N, both> const& i) {
162       s << std::string(i);
163       return s;
164     }
165
166 }
```

G CPHSTL_Branch/Priority-queue-frameworks/Code

G.1 bit-manipulation.h++

```
#include "bit_manipulation.h++" // The underscore redirects to Asgers version.
```

G.2 bit-store.h++

```
/*
  A bit store keeps a sequence of bits in a single word. Requirement:
  The length of the sequence should not be larger than the size of a
  word measured in bits.

  Author: Jyrki Katajainen Â© 2009, 2010
*/

#ifndef __CPHSTL_BIT_STORE___
#define __CPHSTL_BIT_STORE___

#include "assert.h++"
#include "bit-manipulation.h++"
#include <climits> // CHAR_BIT
#include <cstddef>
#include <iostream>

namespace cphstl {

  template <typename W>
  class bit_store;

  template <>
  class bit_store<unsigned long> {
  public:

    typedef bool value_type;
    typedef unsigned long word_type;
    typedef std::size_t size_type;

    enum {word_size = CHAR_BIT * sizeof(word_type)};

    explicit bit_store(word_type value = 0)
      : word(value) {
    }

    operator word_type() const {
      return word;
    }

    size_type size() const {
      return population_count(word);
    }

    size_type capacity() const {
      return word_size;
    };

    template <typename I>
    void set(I index) {
      assert(word_type(index) < word_size);
      word_type mask = word_type(1) << word_type(index);
      word &= ~ mask;
      word |= mask;
    }

    template <typename I>
    void unset(I index) {
      assert(word_type(index) < word_size);
      word_type mask = word_type(1) << word_type(index);
      word &= ~ mask;
    }

```

```
64      template <typename I>
65      bool get(I index) const {
66        assert(index < word_size);
67        word_type v = word_type(1) << index;
68        v = word & v;
69        return (v > 0);
70      }
71
72      size_type least_significant_one() const {
73        assert(size() != 0);
74        return trailing_zeros(word);
75      }
76
77      size_type most_significant_one() const {
78        assert(size() != 0);
79        return capacity() - leading_zeros(word) - 1;
80      }
81
82      size_type choose() const {
83        assert(size() != 0);
84        return most_significant_one();
85      }
86
87    protected:
88
89      word_type word;
90    };
91
92    template <>
93    class bit_store<unsigned long long> {
94    public:
95
96      typedef bool value_type;
97      typedef unsigned long long word_type;
98      typedef std::size_t size_type;
99
100     enum {word_size = CHAR_BIT * sizeof(word_type)};
101
102     explicit bit_store(word_type value = 0)
103       : word(value) {
104     }
105
106     operator word_type() const {
107       return word;
108     }
109
110     size_type size() const {
111       return population_count(word);
112     }
113
114     size_type capacity() const {
115       return word_size;
116     };
117
118     template <typename I>
119     void set(I index) {
120       assert(word_type(index) < word_size);
121       word_type mask = word_type(1) << word_type(index);
122       word &= ~ mask;
123       word |= mask;
124     }
125
126     template <typename I>
127     void unset(I index) {
128       assert(word_type(index) < word_size);
129       word_type mask = word_type(1) << word_type(index);
130       word &= ~ mask;
131     }
132
133     template <typename I>
134     bool get(I index) const {
135       assert(index < word_size);
136       word_type v = word_type(1) << index;
137       v = word & v;
```

```
138         return (v > 0);
139       }
140
141       int least_significant_one() const {
142         assert(size() != 0);
143         return trailing_zeros(word);
144       }
145
146       int most_significant_one() const {
147         assert(size() != 0);
148         return capacity() - leading_zeros(word) - 1;
149       }
150
151       int choose() const {
152         assert(size() != 0);
153         return most_significant_one();
154       }
155
156     protected:
157
158       word_type word;
159     };
160   }
161
162   #if defined(UNITTEST_BIT_STORE)
163   #include "assert.h++"
164   #include <iostream>
165
166   template <typename T>
167   class unittest_bit_store {
168   public:
169
170     void operator()() {
171       cphstl::bit_store<T> word;
172       T n = word; // convertion operator
173       assert(n == 0);
174       T capacity = word.capacity(); // capacity
175       assert(capacity == sizeof(T) * CHAR_BIT);
176       word.set(0);
177       word.set(0);
178       assert(word == 1); // set
179       bool bit_zero = word.get(0);
180       assert(bit_zero == 1); // get
181       bool bit_one = word.get(1);
182       assert(bit_one == 0);
183       bool bit_two = word.get(2);
184       assert(bit_two == 0);
185       word.set(1);
186       assert(word == 3);
187       assert(word.size() == 2); // size
188       std::size_t i = word.choose();
189       assert(i == 1);
190       word.set(1);
191       assert(word.size() == 2);
192       word.unset(1); // unset
193       word.unset(1); // unset
194       assert(word.size() == 1);
195       i = word.choose();
196       assert(i == 0);
197     }
198   };
199
200   int main(int, char**) {
201     unittest_bit_store<unsigned long> s;
202     s();
203     unittest_bit_store<unsigned long long> t;
204     t();
205     return 0;
206   }
207
208   #endif
209   #endif
```

G.3 blank-mark-store.h++

```
/*
  A blank mark store; a mark must be removed before any new marks are
  introduced.

  Author: Jyrki Katajainen Â© 2009
*/

#ifndef _CPHSTL_BLANK_MARK_STORE_
#define _CPHSTL_BLANK_MARK_STORE_

#include <algorithm> // std::swap
#include "assert.h++"
#include "comparator-proxy.h++"
#include <cstddef> // std::size_t
#include <iostream>

namespace cphstl {

  template <typename C, typename A, typename E>
  class blank_mark_store {
  public:

    typedef C comparator_type;
    typedef A allocator_type;
    typedef E encapsulator_type;
    typedef std::size_t size_type;

  protected:

    comparator_proxy<C> comparator;
    E* single_mark;

  private:

    blank_mark_store(blank_mark_store const&);
    blank_mark_store& operator = (blank_mark_store const&);

  public:

    blank_mark_store(C const& c = C(), A const& = A())
      : comparator(c), single_mark(0) {
    }

    ~blank_mark_store() {
    }

    size_type footprint(size_type) const {
      return sizeof(blank_mark_store);
    }

    E* find_top() const {
      return single_mark;
    }

    bool is_marked(E const* p) const {
      return single_mark == p;
    }

    void mark(E* p) {
      assert(single_mark == 0);
      single_mark = p;
    }

    void unmark(E* p) {
      single_mark = (single_mark == p) ? 0 : single_mark;
    }

    template <typename H>
    void reduce(H& heap_store) {
      if (single_mark == 0) {
  return;
```

```
72          }
73          E* q = single_mark;
74          E* p = (*q).distinguished_ancestor();
75          while (p != 0) {
76    if (comparator((*p).element(), (*q).element())) {
77
78  #ifdef FIRST_VERSION
79
80        (*q).swap_nodes(p);
81
82  #else
83
84        q = (*q).promote(p);
85
86  #endif
87
88        p = (*q).distinguished_ancestor();
89    }
90    else {
91      break;
92    }
93          }
94          if ((*q).is_root()) {
95    heap_store.replace(q);
96          }
97          single_mark = 0;
98      }
99
100     template <typename H>
101     void meld(blank_mark_store& another_mark_store, H& heap_store) {
102       if (another_mark_store.single_mark != 0) {
103   another_mark_store.reduce(heap_store);
104       }
105     }
106
107     void swap(blank_mark_store& another_mark_store) {
108       // Precondition: The comparators are compatible and not swapped.
109       std::swap(single_mark, another_mark_store.single_mark);
110     }
111
112     void cleaning_transformation(E*) {
113     }
114
115     E* sibling_transformation(E*) {
116       return 0;
117     }
118
119 #ifdef DEBUG
120
121     bool is_valid() {
122       return true;
123     }
124
125 #endif
126
127   };
128 }
129
130 #endif
```

G.4 eager-mark-store.h++

```
1  /*
2    An eager mark store
3
4    Author: Stefan Edelkamp Â© 2009
5  */
6
7  #ifndef _CPHSTL_EAGER_MARK_STORE_
8  #define _CPHSTL_EAGER_MARK_STORE_
9
10 #include "assert.h++"
11 #include "bit-store.h++"
```

```
12 #include "lazy-mark-store.h++"
13 #include <cstddef> // std::size_t
14 #include <cmath> // ilogb
15 #include <iostream>
16 #include <vector>
17
18 extern int ilogb(double) throw();
19
20 namespace cphstl {
21
22   template <typename C, typename A, typename E>
23   class eager_mark_store: public lazy_mark_store<C,A,E> {
24   public:
25
26     typedef C comparator_type;
27     typedef A allocator_type;
28     typedef E encapsulator_type;
29     typedef typename E::mark_type mark_type;
30     typedef typename E::height_type height_type;
31     typedef typename E::index_type index_type;
32     typedef unsigned long word_type;
33     typedef std::size_t size_type;
34
35     typedef eager_mark_store<C, A, E> M;
36     typedef lazy_mark_store<C, A, E> B;
37
38     struct entry {
39       encapsulator_type* position[3];
40       word_type matesize;
41     };
42
43   protected:
44
45     std::vector<entry*> entries;
46
47     B const* const up_cast(M const* const d) const {
48       return static_cast<B const* const>(d);
49     }
50
51     B* const up_cast(M* const d) const {
52       return static_cast<B* const>(d);
53     }
54
55   private:
56
57     eager_mark_store(eager_mark_store const&);
58     eager_mark_store& operator = (eager_mark_store const&);
59
60   public:
61
62     eager_mark_store(C const& c = C(), A const& a = A())
63       : lazy_mark_store<C,A,E>(c,a) {
64     }
65
66     ~eager_mark_store() {
67     }
68
69 #ifdef DEBUG
70
71     bool is_valid() {
72       return true;
73     }
74
75     void print() {
76       for(height_type h = 0; h < entries.size(); h++) {
77   entry* e1 = entries[h];
78   std::cout << "␣(" << h << ")"
79       << e1->matesize
80       << "[";
81   std::cout << e1->position[0] << "/";
82   if (e1->position[0])
83     std::cout << e1->position[0]->type();
84   std::cout << ",";
85
```

```
86      std::cout << e1->position[1] << "/";
87      if (e1->position[1])
88        std::cout << e1->position[1]->type();
89      std::cout << ",";
90
91      std::cout << e1->position[2] << "/";
92      if (e1->position[2])
93        std::cout << e1->position[2]->type();
94      std::cout << ",";
95          }
96          std::cout << "␣teams␣" << (*this).teams;
97          std::cout << "␣runs␣"  << (*this).runs;
98      std::cout << "\n";
99        }
100
101   #endif
102
103       size_type footprint(size_type n) const {
104         return
105     (ilogb(n) + 1) * (4 * sizeof(E*) +
106           2 * sizeof(word_type)) +
107     2 * sizeof(word_type) +
108     2 * sizeof(word_type);
109       }
110
111       E* find_top() const {
112         if ((*this).runs || (*this).teams) {
113     E* top = 0;
114     for(height_type h = 0; h < entries.size(); h++) {
115       entry* e = entries[h];
116       if (top == 0) {
117         if (e->matesize > 1)
118           top = e->position[1];
119       }
120       else {
121         if(e->matesize == 2 &&
122            comparator(top->element(), e->position[1]->element()))
123           top = e->position[1];
124       }
125     }
126     return top;
127         }
128         else {
129     return 0;
130         }
131       }
132
133       template <typename H>
134       void reduce(H& heap_store) {
135         // std::cout << " reduce called "<< std::endl;
136         //  print();
137         assert(is_valid());
138         while ((*this).runs.size() || (*this).teams.size()) {
139     if ((*this).teams.size()) {
140        // std::cout << " case teams "<< std::endl;
141       word_type m = (*this).teams.choose();
142        // std::cout << " height " << m << std::endl;
143       E* p = entries[m]->position[1];
144       assert((*p).type() == E::singleton);
145       E* q = entries[m]->position[2];
146       assert((*q).type() == E::singleton);
147
148        // std::cout << " before single transform "<< std::endl;
149       singleton_transformation(p, q, heap_store);
150        // std::cout << " after single transform "<< std::endl;
151     }
152     else { // ((*this).runs) {
153        // std::cout << " case runs "<< std::endl;
154       word_type m = (*this).runs.choose();
155       E* r = entries[m]->position[0];
156       //       entries[m]->position[0]->type() == E::leader ?
157       //       entries[m]->position[0] : entries[m]->position[1] ;
158       assert((*r).type() == E::leader);
159        // std::cout << " before run transform "<< std::endl;
```

```
160       run_transformation(r, heap_store);
161        // std::cout << " after run transform "<< std::endl;
162     }
163         }
164          // std::cout << " end reduce "<< std::endl;
165          //       print();
166 
167 #ifndef NDEBUG
168 
169         for(height_type h = 0; h < entries.size(); h++) {
170     entry* e1 = entries[h];
171 
172     if (e1->position[0]) exit(1);
173     if (e1->position[1] && e1->position[2]) exit(1);
174         }
175 
176 #endif
177 
178       }
179 
180       template <typename H>
181       void meld(eager_mark_store& other, H& heap_store) {
182 
183         for (height_type h = 0; h < other.entries.size(); h++) {
184 #ifdef DEBUG
185     entry* e1 = entries[h];
186 #endif
187     entry* e2 = other.entries[h];
188 
189     assert(e1->matesize < 4 && e2->matesize < 4);
190 
191     if (e2->matesize == 2) {
192       E* p = e2->position[1];
193       other.remove_mark(p);
194       other.remove_node(p);
195       insert_node(p);
196       insert_mark(p);
197       reduce(heap_store);
198     }
199         }
200 
201         other.entries.resize(0); // "remove" all elements
202 
203       }
204 
205       void swap(eager_mark_store& other) {
206         // Precondition: The comparators are compatible.
207         std::swap(entries, other.entries);
208         std::swap((*this).runs, other.runs);
209         // std::swap(singles, other.singles);
210         std::swap((*this).teams, other.teams);
211       }
212 
213     protected:
214 
215       void insert_node(E* q) {
216         // entries.push_back(q);
217         // std::cout << "inserting node " << q << ": " << entries.size() << "/" << (*q).
                height() + 1 << std::endl;
218 
219         if (height_type((*q).height() + 1) > entries.size()) {
220     height_type max = std::max(entries.size(), size_type((*q).height() + 1));
221     // // std::cout << "expanding from depth " << entries.size() << " to depth " << max <<
            std::endl;
222     height_type i = entries.size();
223     for (; i < max; i++) {
224       entry* e = new entry;
225       e->position[0] = e->position[1] = e->position[2] = 0;
226       e->matesize = 1;
227       entries.push_back(e) ;
228     }
229         }
230 
231         height_type nheight = q->height();
```

```
232         entry* e = entries[nheight];
233
234         switch ((*q).type()) {
235
236         case E::member:
237     // std::cout << "newly insert member " << q << std::endl;
238     e->position[0] = q;
239     break;
240
241         case E::leader:
242     // std::cout << "node leader newly inserted " << std::endl;
243
244     if (nheight) {
245       entry* e1 = entries[nheight-1];
246       if (e1->position[0] == q->left())
247         break;
248       if (q->left()->type() == E::singleton) {
249         // std::cout << "left " << q->left() << " is singleton " << std::endl;
250         if(e1->position[1] == q->left())
251           e1->position[1] = e1->position[2];
252         e1->position[2] = 0;
253         e1->matesize--;
254         (*this).teams.unset(nheight-1);
255
256         e1->position[0] = q->left();
257         q->left()->type() = E::member;
258       }
259     }
260     break;
261
262         default:
263     ;
264         }
265       }
266
267       void insert_mark(E* q) {
268         // std::cout << "insert mark " << q << " type " << q->type() << std::endl;
269         // Precondition: (*q).type() must have the new value
270
271         height_type nheight = q->height();
272         entry* e = entries[nheight];
273         switch ((*q).type()) {
274
275         case E::leader:
276     // std::cout << "insert leader " << q << std::endl;
277
278     e->position[0] = q;
279     (*this).runs.set(nheight);
280     break;
281
282         case E::member:
283     // std::cout << "error member marked " << std::endl;
284     exit(1);
285     break;
286
287         case E::singleton:
288     // std::cout << "insert singleton " << q << std::endl;
289     e->position[0] = 0;
290
291   #ifndef NDEBUG
292     if (e->matesize > 3) {
293       // std::cout << "error mate insert "<< std::endl;
294       exit(1);
295     }
296   #endif
297
298     if (e->matesize > 1) // elect captain
299       (*this).teams.set(nheight);
300
301     e->position[e->matesize++] = q;
302     break;
303
304         default:
305     ;
```

```
306         }
307       }
308
309       void remove_mark(E* r) {
310         // Precondition: (*r).type() must have the old value
311          // std::cout << "removing mark " << r << std::endl;
312         height_type nheight = r->height();
313         entry* e = entries[nheight];
314
315         switch ((*r).type()) {
316         case E::leader:
317
318    e->position[0] = 0;
319    (*this).runs.unset(nheight);
320    break;
321
322         case E::singleton:
323
324  #ifndef NDEBUG
325    if (e->matesize == 1) {
326      // std::cout << "error mate remove "<< std::endl;
327      exit(1);
328    }
329  #endif
330
331    if(e->matesize-- > 1) {
332      (*this).teams.unset(nheight);
333    }
334
335    if(e->position[1] == r)
336      e->position[1] = e->position[2];
337    e->position[2] = 0;
338
339    break;
340         default:
341    ;
342         }
343         r->type() = E::unmarked;
344       }
345
346       void remove_node(E* x) {
347          // std::cout << "removing node " << x << ": " << entries.size() << "/" << (*x).
                height() + 1 << std::endl;
348
349         height_type nheight = x->height();
350         entry* e = entries[nheight];
351
352         switch ((*x).type()) {
353         case E::member:
354    e->position[0] = 0;
355    break;
356
357         default:
358    ;
359         }
360
361         // std::cout << " end of remove node " << std::endl;
362
363         /*
364         height_type max =
365    std::max(entries.size(), height_type((*x).height() + 1));
366         if (max == entries.size()) {
367    entry* e = entries.back();
368    entries.pop_back();
369    delete e;
370
371    contract(); // not implemented in yirkis code
372         }
373         */
374         return;
375         assert(is_valid());
376       }
377    };
378  }
```

```
379
380 #endif
```

G.5 element-encapsulator.h++

```
1  /*
2    An element encapsulator; each encapsulator knows the position in the
3    data structure where it is stored.
4
5    Author: Jyrki Katajainen Â© 2009, 2010
6  */
7
8  #ifndef _CPHSTL_ELEMENT_ENCAPSULATOR_
9  #define _CPHSTL_ELEMENT_ENCAPSULATOR_
10
11 #include <cstdlib>
12
13 namespace cphstl {
14
15   template <typename V, typename P, typename A>
16   class element_encapsulator {
17   public:
18
19     typedef V value_type;
20     typedef P position_type;
21     typedef A allocator_type;
22
23     value_type value_;
24     position_type position_;
25
26     explicit element_encapsulator (value_type v, allocator_type const& = A()) :
27       value_(v), position_(0) {
28     }
29
30     V const& element() const {
31       return value_;
32     }
33
34     V& element() {
35       return value_;
36     }
37
38     position_type position() const {
39       return position_;
40     }
41
42     position_type& position() {
43       return position_;
44     }
45   };
46 }
47
48 #endif
```

G.6 fat-weak-heap-node.h++

```
1  /*
2    A node used by a perfect weak heap in a run-relaxed weak queue
3
4    Authors: Jyrki Katajainen, Jens Rasmussen Â© 2008, 2009
5  */
6
7  #ifndef _CPHSTL_FAT_WEAK_HEAP_NODE_
8  #define _CPHSTL_FAT_WEAK_HEAP_NODE_
9
10 #include <algorithm> // std::swap
11 #include "assert.h++"
12 #include "heap-node.h++"
13
14 namespace cphstl {
15
```

```
16    template <typename V, typename A>
17    class fat_weak_heap_node
18      : public heap_node<V, A, fat_weak_heap_node<V, A> > {
19    public:
20
21      typedef unsigned char height_type;
22      typedef signed char index_type;
23      enum mark_type {unmarked = 0, member, leader, singleton};
24      typedef fat_weak_heap_node<V, A> N;
25      typedef heap_node<V, A, N> B;
26
27    protected:
28
29      height_type height_;
30      index_type index_;
31      mark_type type_;
32
33      B const* const up_cast(N const* const d) const {
34        return static_cast<B const* const>(d);
35      }
36
37      B* const up_cast(N* const d) const {
38        return static_cast<B* const>(d);
39      }
40
41      B* const up_cast(N* const d) {
42        return static_cast<B* const>(d);
43      }
44
45    public:
46
47      fat_weak_heap_node(V const& v, A const& a)
48        : heap_node<V, A, N>(v, a), height_(0), index_(-1),
49    type_(unmarked) {
50      }
51
52      height_type height() const {
53        assert(this != 0);
54        return height_;
55      }
56
57      height_type& height() {
58        assert(this != 0);
59        return height_;
60      }
61
62      index_type index() const {
63        assert(this != 0);
64        return index_;
65      }
66
67      index_type& index() {
68        assert(this != 0);
69        return index_;
70      }
71
72      mark_type type() const {
73        assert(this != 0);
74        return type_;
75      }
76
77      mark_type& type() {
78        assert(this != 0);
79        return type_;
80      }
81
82      template <typename C>
83      N* basic_join(N* q, C const& comparator) {
84        N* p = this;
85        if (comparator((*p).element(), (*q).element())) {
86          N* c = (*q).right();
87          if (c != 0) {
88            (*c).parent() = p;
89          }
```

```
90          (*p).left() = c;
91          (*q).right() = p;
92          (*p).parent() = q;
93      (*q).height() += 1;
94          return q;
95        }
96        else {
97          N* c = (*p).right();
98          if (c != 0) {
99            (*c).parent() = q;
100         }
101         (*q).left() = c;
102         (*p).right() = q;
103         (*q).parent() = p;
104     (*p).height() += 1;
105         return p;
106       }
107     }
108
109     template <typename C, typename M>
110     N* join(N* q, C const& comparator, M& mark_store) {
111       N* p = this;
112       assert(p != 0);
113       assert(q != 0);
114       assert((*p).type() == N::unmarked);
115       assert((*q).type() == N::unmarked);
116       B* u = up_cast(p);
117       N* r = (*u).join(q, comparator, mark_store);
118       (*r).height() += 1;
119       N* a = (*r).right() -> left();
120       N* b = (*r).right() -> right();
121       bool a_unmarked = a == 0 || (*a).type() == N::unmarked;
122       bool b_unmarked = b == 0 || (*b).type() == N::unmarked;
123       int item = 2 * a_unmarked + b_unmarked;
124       switch (item) {
125       case 0 /* marked marked */:
126     (void) mark_store.sibling_transformation(a);
127     break;
128       case 1 /* marked unmarked */:
129     mark_store.cleaning_transformation(a);
130     break;
131       case 2 /* unmarked marked */:
132     break;
133       case 3 /* unmarked unmarked */:
134     break;
135       default:
136     assert(false);
137       }
138       return r;
139     }
140
141     template <typename C, typename M>
142     N* fast_join(N* q, N*, C const& comparator, M& mark_store) {
143       N* p = this;
144       assert(p != 0);
145       assert(q != 0);
146       assert((*p).type() == N::unmarked);
147       assert((*q).type() == N::unmarked);
148       assert((*p).right() == 0 || (*p).right() -> type() == N::unmarked);
149       if (comparator((*p).element(), (*q).element())) {
150         N* b = (*q).right();
151     bool cleaning_necessary = false;
152         if (b != 0) {
153           (*b).parent() = p;
154     if ((*b).type() != N::unmarked) {
155       cleaning_necessary = true;
156     }
157     }
158         (*p).left() = b;
159         (*q).right() = p;
160         (*p).parent() = q;
161     (*q).height() += 1;
162     if (cleaning_necessary) {
163       mark_store.cleaning_transformation(b);
```

```
164     }
165           return q;
166         }
167         else {
168           N* a = (*p).right();
169           if (a != 0) {
170             (*a).parent() = q;
171           }
172           (*q).left() = a;
173           (*p).right() = q;
174           (*q).parent() = p;
175     (*p).height() += 1;
176           return p;
177         }
178       }
179
180       template <typename C>
181       N* split(C const& comparator) {
182         assert((*this).height() > 0);
183         B* b = up_cast(this);
184         N* p = (*b).split(comparator);
185         (*this).height() -= 1;
186         return p;
187       }
188
189       void swap_neighbours(N* r) {
190         N* p = this;
191         assert(p != 0);
192         assert(r == (*p).right());
193         N* a = (*p).left();
194         N* b = (*r).left();
195         N* c = (*r).right();
196         (*r).left() = a;
197         if (a != 0 && (! (*p).is_root())) {
198     (*a).parent() = r;
199         }
200         (*r).right() = p;
201         (*p).parent() = r;
202         (*p).left() = c;
203         (*p).right() = b;
204         if (b != 0) {
205     (*c).parent() = p; // be careful!
206     (*b).parent() = p;
207         }
208         (*r).height() += 1;
209         (*p).height() -= 1;
210       }
211
212       void swap_roots(N* q) {
213         B* b = up_cast(this);
214         (*b).swap_roots(q);
215         std::swap((*this).height(), (*q).height());
216       }
217
218       N* release_root() {
219         B* b = up_cast(this);
220         N* s = (*b).release_root();
221         (*this).height() = 0;
222         return s;
223       }
224
225       N* promote(N* p) {
226         B* b = up_cast(this);
227         N* q = (*b).promote(p);
228         std::swap((*this).height(), (*p).height());
229         return q;
230       }
231
232   #ifdef DEBUG
233
234       void show_tree() const {
235         N const* t = this;
236         std::list<N const*> level;
237         level.push_front(t);
```

```
238         while (! level.empty()) {
239           typename std::list<N const*>::iterator last = level.end();
240           --last;
241           typename std::list<N const*>::iterator p = level.begin();
242           bool stop = false;
243           while (! stop) {
244             N const* t = *p;
245             if (p == last) {
246         stop = true;
247       }
248             ++p;
249             level.pop_front();
250             if (! (*t).is_root() && (*t).left() != 0) {
251               level.push_back((*t).left());
252             }
253             if ((*t).right() != 0) {
254               level.push_back((*t).right());
255             }
256             std::cout << "(" << (*t).element() <<
257         ",␣" << int((*t).height()) <<
258         ",␣" << int((*t).index()) <<
259         ",␣" << (*t).type() << ")␣";
260           }
261           std::cout << "\n";
262           std::cout.flush();
263         }
264       }
265
266       template <typename C, typename M>
267       bool is_valid(C const& comparator, M const& mark_store) const {
268         N const* t = this;
269         bool valid = true;
270         if ((*t).parent() != 0) {
271           valid &= t -> parent() -> left() == t ||
272             t -> parent() -> right() == t;
273     valid &= t -> parent() -> height() == t -> height() + 1;
274     if (! valid) std::cout << "error:␣parent\n";
275         }
276         if ((*t).left() != 0) {
277           valid &= t -> left() -> parent() == t;
278     if (! valid) std::cout << "error:␣left\n";
279         }
280         if ((*t).right() != 0) {
281           valid &= t -> right() -> parent() == t;
282     if (! valid) std::cout << "error:␣right\n";
283         }
284         if (! (*t).is_root() && ! mark_store.is_marked(t)) {
285           valid &= ! comparator((*t).distinguished_ancestor() -> element(), (*t).element());
286     if (! valid) std::cout << "error:␣half␣order\n";
287         }
288         if ((*t).is_root()) {
289     valid &= (*t).type() == N::unmarked;
290     assert((*t).parent() == 0);
291     if (! valid) std::cout << "error:␣root\n";
292         }
293         else {
294       N const* p = (*t).parent();
295       N const* r = (*t).left();
296       if ((*p).left() == t) {
297         if (mark_store.is_marked(p)) {
298           if (mark_store.is_marked(t)) {
299             valid &= (*t).type() == N::member;
300             if (! valid) std::cout << "error:␣member\n";
301           }
302         }
303         else {
304           if (mark_store.is_marked(t)) {
305             if (r != 0 && mark_store.is_marked(r)) {
306         valid &= (*t).type() == N::leader;
307         if (! valid) std::cout << "error:␣leader␣(left)\n";
308             }
309             else {
310         valid &= (*t).type() == N::singleton;
311         if (! valid) std::cout << "error:␣singleton␣(left)\n";
```

```
312           }
313         }
314       }
315     }
316     else {
317       if (r != 0 && mark_store.is_marked(r)) {
318         if (mark_store.is_marked(t)) {
319           valid &= (*t).type() == N::leader;
320           if (! valid) std::cout << "error: leader (right)\n";
321         }
322       }
323       else {
324         if (mark_store.is_marked(t)) {
325           valid &= (*t).type() == N::singleton;
326           if (! valid) std::cout << "error: singleton (right)\n";
327         }
328       }
329     }
330         }
331         return valid;
332       }
333
334 #endif
335
336   };
337 }
338
339 #endif
```

G.7 heap-node.h++

```
1  /*
2    A heap node used as a base for various specialized heap nodes
3
4    Authors: Asger Bruun, Jyrki Katajainen Â© 2009, 2010
5  */
6
7  #ifndef _CPHSTL_HEAP_NODE_
8  #define _CPHSTL_HEAP_NODE_
9
10 #include "assert.h++"
11 #include <cstddef> // std::size_t
12 #include <iostream>
13 #include <list>
14
15 namespace cphstl {
16
17   template <typename V, typename A, typename N>
18   class heap_node {
19   public:
20
21     typedef V value_type;
22     typedef A allocator_type;
23     typedef std::size_t size_type;
24     typedef unsigned char height_type;
25     typedef heap_node<V, A, N> self_type;
26
27     struct hole_type {
28       N* parent;
29       N* current;
30       union {
31   N* left;
32   void* owner;
33       };
34
35       hole_type(N* p)
36         : parent((*p).parent()), current(p) {
37   if (parent != 0) {
38     left = (*p).left();
39   }
40   else {
41     owner = (*p).owner();
42   }
```

```
43        }
44      };
45
46      N* parent_;
47      union {
48        N* left_;
49        void* owner_;
50      };
51      N* right_;
52      V value_;
53
54    private:
55
56      heap_node();
57      heap_node(heap_node const&);
58      heap_node& operator = (heap_node const&);
59
60    protected:
61
62      N const* const down_cast(self_type const* const b) const {
63        return static_cast<N const* const>(b);
64      }
65
66      N* const down_cast(self_type* const b) const {
67        return static_cast<N* const>(b);
68      }
69
70      N* const down_cast(self_type* const b) {
71        return static_cast<N* const>(b);
72      }
73
74    public:
75
76      heap_node(V const& v, A const&)
77        : parent_(down_cast(0)), left_(down_cast(0)), right_(down_cast(0)),
78        value_(v) {
79      }
80
81      static size_type footprint() {
82        return sizeof(N);
83      }
84
85      bool is_root() const {
86        return parent_ == 0;
87      }
88
89      bool is_leaf() const {
90        return right_ == 0;
91      }
92
93      V const& element() const {
94        return value_;
95      }
96
97      V& element() {
98        return value_;
99      }
100
101     N* left() const {
102       return left_;
103     }
104
105     N*& left() {
106       return left_;
107     }
108
109     N* right() const {
110       return right_;
111     }
112
113     N*& right() {
114       return right_;
115     }
116
```

```
117       N* parent() const {
118         return parent_;
119       }
120
121       N*& parent() {
122         return parent_;
123       }
124
125       void* owner() const {
126         return owner_;
127       }
128
129       void*& owner() {
130         return owner_;
131       }
132
133       template <typename C, typename M>
134       N* join(N* q, C const& comparator, M&) {
135         N* p = down_cast(this);
136         if (comparator((*p).element(), (*q).element())) {
137           N* c = (*q).right();
138           if (c != 0) {
139             (*c).parent() = p;
140           }
141           (*p).left() = c;
142           (*q).right() = p;
143           (*p).parent() = q;
144           return q;
145         }
146         else {
147           N* c = (*p).right();
148           if (c != 0) {
149             (*c).parent() = q;
150           }
151           (*q).left() = c;
152           (*p).right() = q;
153           (*q).parent() = p;
154           return p;
155         }
156       }
157
158       template <typename C, typename M>
159       N* fast_join(N* q, N*, C const& comparator, M& mark_store) {
160         N* p = down_cast(this);
161         return (*p).join(q, comparator, mark_store);
162       }
163
164       template <typename C>
165       N* split(C const&) {
166         N* p = down_cast(this);
167         assert(p != 0);
168         assert((*p).right() != 0);
169         N* q = (*p).right();
170         N* r = (*q).left();
171         (*p).right() = r;
172         if (r != 0) {
173           (*r).parent() = p;
174         }
175         (*q).parent() = 0;
176         (*q).left() = 0;
177         return q;
178       }
179
180       void swap_roots(N* q) {
181         N* p = down_cast(this);
182         assert((*p).is_root());
183         assert((*q).is_root());
184         N* c = (*p).right();
185         N* g = (*q).right();
186         (*p).right() = g;
187         (*q).right() = c;
188         if (c != 0) {
189           (*c).parent() = q;
190         }
```

```
191         if (g != 0) {
192           (*g).parent() = p;
193         }
194       }
195 
196       N* release_root() {
197         N* p = down_cast(this);
198         N* q = (*p). right();
199         (*p).right() = 0;
200         (*q).parent() = 0;
201         return q;
202       }
203 
204       N* release_subheap() {
205         N* p = down_cast(this);
206         N* q = (*p).left();
207         (*p).left() = 0;
208         if (q != 0) {
209     (*q).parent() = 0;
210         }
211         return q;
212       }
213 
214       hole_type splice_out() {
215         // Note: This function leaves the underlying tree broken,
216         // untraversable, and unprintable until splice_in.
217         assert(this != 0);
218         hole_type hole(down_cast(this));
219         (*this).parent() = 0;
220         (*this).left() = 0;
221         return hole;
222       }
223 
224       template <typename H>
225       void splice_in(hole_type& hole, H& heap_store) {
226         assert(hole.current != 0);
227         N* p = down_cast(this);
228         (*p).parent() = hole.parent;
229         if (hole.parent != 0 && (*hole.parent).left() == hole.current) {
230           (*hole.parent).left() = p;
231         }
232         if (hole.parent != 0 && (*hole.parent).right() == hole.current) {
233           (*hole.parent).right() = p;
234         }
235         if (hole.parent == 0) {
236     typedef typename H::heap_proxy_type heap_proxy_type;
237     (*p).owner() = hole.owner;
238     heap_store.replace(p);
239         }
240         else {
241     (*this).left() = hole.left;
242     if (hole.left != 0) {
243       (*hole.left).parent() = p;
244     }
245         }
246       }
247 
248   #ifdef FIRST_VERSION
249 
250       void swap_nodes(N* q) {
251         // Warning: The backpointers at owners are not corrected because
252         // the type of the owners is not known.
253         N* p = down_cast(this);
254         assert(p != 0);
255         assert(q != 0);
256         N* a = (*p).parent_;
257         N* b = (*p).left_;
258         N* c = (*p).right_;
259         N* e = (*q).parent_;
260         N* f = (*q).left_;
261         N* g = (*q).right_;
262 
263         if (a != 0 && (*a).left_ == p) {
264           (*a).left_ = q;
```

```
265         }
266         if (a != 0 && (*a).right_ == p) {
267           (*a).right_ = q;
268         }
269         if (b != 0 && (! (*p).is_root()) && (*b).parent_ == p) {
270           (*b).parent_ = q;
271         }
272         if (c != 0 && (*c).parent_ == p) {
273           (*c).parent_ = q;
274         }
275         if (e != 0 && (*e).left_ == q) {
276           (*e).left_ = p;
277         }
278         if (e != 0 && (*e).right_ == q) {
279           (*e).right_ = p;
280         }
281         if (f != 0 && (! (*p).is_root()) && (*f).parent_ == q) {
282           (*f).parent_ = p;
283         }
284         if (g != 0 && (*g).parent_ == q) {
285           (*g).parent_ = p;
286         }
287
288         (*p).parent_ = e;
289         (*p).left_ = f;
290         (*p).right_ = g;
291         (*q).parent_ = a;
292         (*q).left_ = b;
293         (*q).right_ = c;
294
295         if (a == q && f == p) {
296           (*p).left_ = q;
297           (*q).parent_ = p;
298           return;
299         }
300         if (a == q && g == p) {
301           (*p).right_ = q;
302           (*q).parent_ = p;
303           return;
304         }
305         if (b == q) {
306           (*p).parent_ = q;
307           (*q).left_ = p;
308           return;
309         }
310         if (c == q) {
311           (*p).parent_ = q;
312           (*q).right_ = p;
313           return;
314         }
315       }
316
317   #endif
318
319       template <typename C>
320       N* distinguished_descendant(C const&) const {
321         N const* q = down_cast(this);
322         assert(q != 0);
323         return ((*q).is_root())? (*q).right() : (*q).left();
324       }
325
326       N* distinguished_ancestor() const {
327         N const* q = down_cast(this);
328         assert(q != 0);
329         N* p = (*q).parent();
330         while (p != 0 && (*p).left() == q) {
331           q = p;
332           p = (*p).parent();
333         }
334         return p;
335       }
336
337       N* promote(N* p) {
338         N* q = down_cast(this);
```

```
339         assert(p == (*q).distinguished_ancestor());
340         if (p == (*q).parent()) {
341           assert((*p).right() == q);
342           N* a = (*p).parent();
343           N* b = (*p).left();
344           N* f = (*q).left();
345           N* g = (*q).right();
346           (*p).parent() = q;
347           (*p).left() = f;
348           (*p).right() = g;
349           (*q).parent() = a;
350           (*q).left() = b;
351           (*q).right() = p;
352           if (a != 0) {
353             if ((*a).right() == p) {
354               (*a).right() = q;
355             }
356             else {
357               if (! (*p).is_root()) {
358                 (*a).left() = q;
359               }
360             }
361           }
362           if (b != 0 && (! (*p).is_root())) {
363             (*b).parent() = q;
364           }
365           if (f != 0) {
366             (*f).parent() = p;
367           }
368           if (g != 0) {
369             (*g).parent() = p;
370           }
371         }
372         else {
373           N* a = (*p).parent();
374           N* b = (*p).left();
375           N* c = (*p).right();
376           N* e = (*q).parent();
377           N* f = (*q).left();
378           N* g = (*q).right();
379           (*p).parent() = e;
380           (*p).left() = f;
381           (*p).right() = g;
382           (*q).parent() = a;
383           (*q).left() = b;
384           (*q).right() = c;
385           if (a != 0) {
386             if ((*a).right() == p) {
387               (*a).right() = q;
388             }
389             else {
390               if (! (*p).is_root()) {
391                 (*a).left() = q;
392               }
393             }
394           }
395           if (b != 0 && (! (*p).is_root())) {
396             (*b).parent() = q;
397           }
398           if (c != 0) {
399             (*c).parent() = q;
400           }
401           if (e != 0 && (*e).left() == q) {
402             (*e).left() = p;
403           }
404           if (e != 0 && (*e).right() == q) {
405             (*e).right() = p;
406           }
407           if (f != 0) {
408             (*f).parent_ = p;
409           }
410           if (g != 0) {
411             (*g).parent_ = p;
412           }
```

```
413         }
414         return q;
415       }
416
417       N const* root() const {
418         N const* p = down_cast(this);
419         assert(p != 0);
420         while (! (*p).is_root()) {
421           p = (*p).parent();
422         }
423         return p;
424       }
425
426       N const* successor() const {
427         N const* x = down_cast(this);
428         assert(x != 0);
429         if ((*x).right() != 0) {
430           x = (*x).right();
431           while ((*x).left() != 0) {
432             x = (*x).left();
433           }
434           return x;
435         }
436         N const* y = (*x).parent();
437         while (y != 0 && x == (*y).right()) {
438           x = y;
439           y = (*y).parent();
440         }
441         return y;
442       }
443
444       height_type height() const {
445         N const* p = down_cast(this);
446         assert(p != 0);
447         height_type h = 0;
448         while (! (*p).is_leaf()) {
449           p = (*p).right();
450     h += 1;
451         }
452         return h;
453       }
454
455   #ifdef DEBUG
456
457       void show_tree() const {
458         N const* t = down_cast(this);
459         std::cout << (*t).element() << "␣";
460         std::cout << "\n";
461         std::cout.flush();
462         t = (*t).right();
463         if (t == 0) {
464     return;
465         }
466         std::list<N const*> level;
467         level.push_front(t);
468         while (! level.empty()) {
469           typename std::list<N const*>::iterator last = level.end();
470           --last;
471           typename std::list<N const*>::iterator p = level.begin();
472           bool stop = false;
473           while (! stop) {
474             N const* t = *p;
475             if (p == last) stop = true;
476             ++p;
477             level.pop_front();
478             if ((*t).right() != 0) {
479               level.push_back((*t).left());
480               level.push_back((*t).right());
481             }
482             std::cout << (*t).element() << "␣";
483           }
484           std::cout << "\n";
485           std::cout.flush();
486         }
```

```
487       }
488
489   #endif
490
491     };
492   }
493
494   #endif
```

G.8 heap-proxy.h++

```
1   /*
2     A heap proxy maintains the height and a pointer to the root
3     of each heap.
4
5     Author: Jyrki Katajainen Â© 2009
6   */
7
8   #ifndef _CPHSTL_HEAP_PROXY_
9   #define _CPHSTL_HEAP_PROXY_
10
11  #include "assert.h++"
12  #include <cstddef> // std::size_t
13
14  namespace cphstl {
15
16    template <typename E>
17    class heap_proxy {
18    public:
19
20      typedef E encapsulator_type;
21      typedef std::size_t size_type;
22
23      heap_proxy(E* root, size_type height = 0,
24            heap_proxy* next = 0,
25            heap_proxy* next_pair = 0)
26        : root_(root), height_(height), next_(next), next_pair_(next_pair) {
27      }
28
29      ~heap_proxy() {
30      }
31
32      void update(E* root, size_type height, heap_proxy* next,
33      heap_proxy* next_pair) {
34        assert(this != 0);
35        assert((*root).is_root());
36        root_ = root;
37        height_ = height;
38        next_ = next;
39        next_pair_ = next_pair;
40        (*root).owner() = this;
41      }
42
43      E* root() const {
44        return root_;
45      }
46
47      E*& root() {
48        return root_;
49      }
50
51      size_type height() const {
52        return height_;
53      }
54
55      size_type& height() {
56        return height_;
57      }
58
59      heap_proxy* successor() const {
60        return next_;
61      }
62
```

```
63      heap_proxy*& successor() {
64        return next_;
65      }
66
67      heap_proxy* successor_pair() const {
68        return next_pair_;
69      }
70
71      heap_proxy*& successor_pair() {
72        return next_pair_;
73      }
74
75    protected:
76
77      E* root_;
78      size_type height_;
79      heap_proxy* next_;
80      heap_proxy* next_pair_;
81
82    private:
83
84      heap_proxy();
85      heap_proxy(heap_proxy const&);
86      heap_proxy& operator = (heap_proxy const&);
87
88    };
89  }
90
91  #if defined(UNITTEST_HEAP_PROXY)
92
93  #include <memory> // std::allocator
94  #include "weak-heap-node.h++"
95
96  template <typename T>
97  void test_heap_proxy() {
98    typedef std::allocator<T> A;
99    typedef cphstl::weak_heap_node<T, A> N;
100   typedef cphstl::heap_proxy<N> P;
101   N* dummy = new N(T(0), A());
102   P* p = new P(dummy);
103   P* q = new P(dummy);
104   P* r = new P(dummy);
105
106   (*p).successor() = q;
107   (*q).successor() = r;
108   (*r).successor() = 0;
109
110   assert((*p).height() == 0);
111   assert((*q).root() == dummy);
112   assert((*r).successor_pair() == 0);
113   assert((*p).successor() == q);
114   assert((*q).successor() == r);
115   assert((*r).successor() == 0);
116
117   (*r).height() = 4;
118   assert((*r).height() == 4);
119 }
120
121 int main(int, char**) {
122   test_heap_proxy<int>();
123   test_heap_proxy<char>();
124   return 0;
125 }
126
127 #endif
128 #endif
```

G.9 lazy-mark-store.h++

```
1   /*
2     A lazy mark store
3
4     Author: Jyrki Katajainen Â© 2009, 2010
```

```
5   */
6
7   #ifndef _CPHSTL_LAZY_MARK_STORE_
8   #define _CPHSTL_LAZY_MARK_STORE_
9
10  #include <algorithm> // std::max, std::swap
11  #include "assert.h++"
12  #include "bit-store.h++"
13  #include <cstddef> // std::size_t
14  #include <cmath> // ilogb
15  #include <iostream>
16  #include <vector>
17
18  extern int ilogb(double) throw();
19
20  namespace cphstl {
21
22    template <typename C, typename A, typename E>
23    class lazy_mark_store {
24    public:
25
26      typedef C comparator_type;
27      typedef A allocator_type;
28      typedef E encapsulator_type;
29      typedef typename E::mark_type mark_type;
30      typedef typename E::height_type height_type;
31      typedef typename E::index_type index_type;
32      typedef unsigned long word_type;
33      typedef std::size_t size_type;
34
35    protected:
36
37      C comparator;
38
39      bit_store<word_type> runs;
40      bit_store<word_type> teams;
41
42  #ifndef RANK
43
44      std::vector<E*, A> nodes;
45      std::vector<bit_store<word_type>, A> singles;
46
47  #endif
48
49    private:
50
51      lazy_mark_store(lazy_mark_store const&);
52      lazy_mark_store& operator = (lazy_mark_store const&);
53
54    public:
55
56      lazy_mark_store(C const& c = C(), A const& a = A())
57        : comparator(c), runs(), teams()
58
59  #ifndef RANK
60
61        , nodes(a)
62        , singles(a)
63
64  #endif
65      {
66      }
67
68      ~lazy_mark_store() {
69      }
70
71      size_type footprint(size_type n) const {
72        return (ilogb(n) + 1) * 6 * (sizeof(E*) + 2 * sizeof(word_type)) +
73    2 * sizeof(word_type);
74      }
75
76      E* find_top() const {
77
78  #ifndef RANK
```

```
79
80          if (nodes.size() == 0) {
81    return 0;
82          }
83          E* top = nodes[0];
84          for (index_type i = 1; i < index_type(nodes.size()); ++i) {
85    if (comparator((*top).element(), (*nodes[i]).element())) {
86      top = nodes[i];
87    }
88          }
89          return top;
90
91  #endif
92      }
93
94      bool is_marked(E const* p) const {
95        return (*p).type() != E::unmarked;
96      }
97
98      void mark(E* q) {
99        //        std::cout << " mark called " << q << std::endl;
100       assert(q != 0);
101       if (is_marked(q) || (*q).is_root()) {
102   return;
103         }
104       E* p = (*q).parent();
105       E* r = (*q).left();
106       word_type row = 0;
107       if (q == (*p).left()) {
108   switch ((*p).type()) {
109   case E::unmarked:
110     row = 1;
111     break;
112   case E::member:
113     row = 2;
114     break;
115   case E::singleton:
116     row = 3;
117     break;
118   default:
119     assert(false);
120   }
121       }
122       word_type column = 0;
123       if (r != 0) {
124   switch ((*r).type()) {
125   case E::unmarked:
126     column = 1;
127     break;
128   case E::leader:
129     column = 2;
130     break;
131   case E::singleton:
132     column = 3;
133     break;
134   default:
135     assert(false);
136   }
137       }
138
139       //        std::cout << "-- mark case " << row*4+column << std::endl;
140
141     switch (row * 4 + column) {
142       case 0:
143   mark_action_1(p, q, r);
144   break;
145       case 1:
146   mark_action_1(p, q, r);
147   break;
148       case 2:
149   mark_action_2(p, q, r);
150   break;
151       case 3:
152   mark_action_2(p, q, r);
```

```
153     break;
154         case 4:
155     mark_action_1(p, q, r);
156     break;
157         case 5:
158     mark_action_1(p, q, r);
159     break;
160         case 6:
161     mark_action_2(p, q, r);
162     break;
163         case 7:
164     mark_action_2(p, q, r);
165     break;
166         case 8:
167     mark_action_3(p, q, r);
168     break;
169         case 9:
170     mark_action_3(p, q, r);
171     break;
172         case 10:
173     mark_action_4(p, q, r);
174     break;
175         case 11:
176     mark_action_4(p, q, r);
177     break;
178         case 12:
179     mark_action_5(p, q, r);
180     break;
181         case 13:
182     mark_action_5(p, q, r);
183     break;
184         case 14:
185     mark_action_6(p, q, r);
186     break;
187         case 15:
188     mark_action_6(p, q, r);
189     break;
190         default:
191     assert(false);
192         }
193       }
194
195       void unmark(E* q) {
196         //              std::cout << " unmark called " << q << std::endl;
197         assert(q != 0);
198         E* p = (*q).parent();
199         E* r = (*q).left();
200         E* s;
201         switch ((*q).type()) {
202         case E::unmarked:
203     return;
204         case E::member:
205     if (r == 0 || (*r).type() == E::unmarked) {
206       if ((*p).type() == E::member) {
207         //  std::cout << "-- unmark action " << 1 << std::endl;
208         unmark_action_1(p, q, r);
209       }
210       else {
211         //  std::cout << "-- unmark action " << 2 << std::endl;
212         unmark_action_2(p, q, r);
213       }
214     }
215     else {
216       assert((*r).type() == E::member);
217       E* s = (*r).left();
218       word_type row = word_type((*p).type() == E::leader);
219       word_type column = 0;
220       if (s != 0) {
221         if ((*s).type() == E::unmarked) {
222           column = 1;
223         }
224         else {
225           assert((*s).type() == E::member);
226           column = 2;
```

```
227         }
228       }
229       //  std::cout << "-- unmark case " << row*3+column << std::endl;
230       switch (row * 3 + column) {
231       case 0:
232         unmark_action_3(p, q, r);
233         break;
234       case 1:
235         unmark_action_3(p, q, r);
236         break;
237       case 2:
238         unmark_action_4(p, q, r);
239         break;
240       case 3:
241         unmark_action_5(p, q, r);
242         break;
243       case 4:
244         unmark_action_5(p, q, r);
245         break;
246       case 5:
247         unmark_action_6(p, q, r);
248         break;
249       default:
250         assert(false);
251       }
252     }
253     break;
254         case E::leader:
255     s = (*r).left();
256     if (s == 0) {
257       //   std::cout << "-- unmark action " << 7 << std::endl;
258       unmark_action_7(p, q, r);
259     }
260     else {
261       if ((*s).type() == E::unmarked) {
262         //        std::cout << "-- unmark action " << 7 << std::endl;
263         unmark_action_7(p, q, r);
264       }
265       else {
266         //        std::cout << "-- unmark action " << 8 << std::endl;
267         assert((*s).type() == E::member);
268         unmark_action_8(p, q, r);
269       }
270     }
271     break;
272         case E::singleton:
273     // std::cout << "-- unmark action " << 9 << std::endl;
274     unmark_action_9(p, q, r);
275     break;
276         default:
277     assert(false);
278         }
279           // std::cout << "end unmark  " << std::endl;
280       }
281
282       template <typename H>
283       void reduce(H& heap_store) {
284
285   #ifndef RANK
286         //       std::cout << " base reduce called "<< std::endl;
287         assert(is_valid());
288         if (teams.size() != 0) {
289     height_type h = teams.choose();
290     index_type i = singles[h].choose();
291     E* p = nodes[i];
292     assert((*p).type() == E::singleton);
293     singles[h].unset(i);
294     index_type j = singles[h].choose();
295     singles[h].set(i);
296     E* q = nodes[j];
297     assert((*q).type() == E::singleton);
298     singleton_transformation(p, q, heap_store);
299     return;
300         }
```

```
301         if (runs.size() != 0) {
302   index_type i = runs.choose();
303   E* r = nodes[i];
304   assert((*r).type() == E::leader);
305   run_transformation(r, heap_store);
306         }
307 
308 #endif
309     }
310 
311     template <typename H>
312     void meld(lazy_mark_store& other, H& heap_store) {
313 
314 #ifndef RANK
315 
316     index_type k = other.nodes.size();
317       for (index_type i = 0; i != k; ++i) {
318   E* p = other.nodes[i];
319   other.remove_mark(p);
320   other.remove_node(p);
321   insert_node(p);
322   insert_mark(p);
323   reduce(heap_store);
324       }
325       other.nodes.resize(0);
326       other.singles.resize(0);
327 
328 #endif
329     }
330 
331     void swap(lazy_mark_store& other) {
332 
333 #ifndef RANK
334 
335       // Precondition: The comparators are compatible.
336       std::swap(nodes, other.nodes);
337       std::swap(runs, other.runs);
338       std::swap(singles, other.singles);
339       std::swap(teams, other.teams);
340 
341 #endif
342     }
343 
344 #ifdef DEBUG
345 
346     bool is_valid() const {
347       bool valid = true;
348       for (index_type i = 0; i < index_type(nodes.size()); ++i) {
349   E const* p = nodes[i];
350   valid &= (*p).index() == i;
351   valid &= is_marked(p);
352   if (! valid) {
353     std::cout << "error: nodes\n";
354   }
355       }
356       bit_store<word_type> temp = runs;
357       for (height_type h = 0; h < height_type(temp.size()); ++h) {
358   index_type i = temp.choose();
359   temp.unset(i);
360   E const* p = nodes[i];
361   valid &= (*p).type() == E::leader;
362   if (! valid) {
363     std::cout << "error: runs\n";
364   }
365       }
366       temp = teams;
367       for (height_type j = 0; j < height_type(temp.size()); ++j) {
368   height_type h = temp.choose();
369   temp.unset(h);
370   valid &= singles[h].size() > 1;
371   if (! valid) {
372     std::cout << "error: teams\n";
373   }
374       }
```

```
375         for (height_type h = 0; h < height_type(singles.size()); ++h) {
376     temp = singles[h];
377     for (index_type i = 0; i < index_type(temp.size()); ++i) {
378       index_type j = temp.choose();
379       temp.unset(j);
380       E const* p = nodes[j];
381       valid &= (*p).type() == E::singleton;
382       if (! valid) {
383         std::cout << "error: singles (" << (*p).element() << ")\n";
384       }
385     }
386         }
387 
388         return valid;
389       }
390 
391   #endif
392 
393     protected:
394 
395       virtual void insert_node(E* q) {
396 
397   #ifndef RANK // avoids index manip
398           // std::cout << "inserting base " << "/" << (*q).height() + 1 << std::endl;
399         nodes.push_back(q);
400         size_type max = std::max(singles.size(), size_type((*q).height() + 1));
401         singles.resize(max, bit_store<word_type>());
402         (*q).index() = nodes.size() - 1;
403 
404   #endif
405       }
406 
407       virtual void insert_mark(E* q) {
408 
409   #ifndef RANK // avoids index manip
410 
411         // Precondition: (*q).type() must have the new value
412         switch ((*q).type()) {
413         case E::leader:
414     runs.set((*q).index());
415     break;
416         case E::singleton:
417     singles[(*q).height()].set((*q).index());
418     if (singles[(*q).height()].size() > 1) {
419       teams.set((*q).height());
420     }
421     break;
422         default:
423     ;       }
424 
425   #endif
426       }
427 
428       virtual void remove_mark(E* r) {
429 
430   #ifndef RANK
431         // Precondition: (*r).type() must have the old value
432         switch ((*r).type()) {
433         case E::leader:
434     runs.unset((*r).index());
435     break;
436         case E::singleton:
437     singles[(*r).height()].unset((*r).index());
438     if (singles[(*r).height()].size() < 2) {
439       teams.unset((*r).height());
440     }
441     break;
442         default:
443     ;
444         }
445 
446   #endif
447       }
448 
```

```
449       virtual void remove_node(E* x) {
450
451   #ifndef RANK
452
453         E* y = nodes.back();
454         remove_mark(y);
455         nodes.pop_back();
456         if (x != y) {
457     index_type i = (*x).index();
458     nodes[i] = y;
459     (*y).index() = i;
460     insert_mark(y);
461         }
462         assert(is_valid());
463
464   #endif
465       }
466
467
468       void mark_action_1(E*, E* q, E*) {
469         (*q).type() = E::singleton;
470         insert_node(q);
471         insert_mark(q);
472       }
473
474       void mark_action_2(E*, E* q, E* r) {
475         (*q).type() = E::leader;
476         insert_node(q);
477         insert_mark(q);
478         remove_mark(r);
479         (*r).type() = E::member;
480       }
481
482       void mark_action_3(E*, E* q, E*) {
483         (*q).type() = E::member;
484         insert_node(q);
485       }
486
487       void mark_action_4(E*, E* q, E* r) {
488         (*q).type() = E::member;
489         insert_node(q);
490         remove_mark(r);
491         (*r).type() = E::member;
492       }
493
494       void mark_action_5(E* p, E* q, E*) {
495         remove_mark(p);
496         (*p).type() = E::leader;
497         insert_mark(p);
498         (*q).type() = E::member;
499         insert_node(q);
500       }
501
502       void mark_action_6(E* p, E* q, E* r) {
503         remove_mark(p);
504         (*p).type() = E::leader;
505         insert_mark(p);
506         (*q).type() = E::member;
507         insert_node(q);
508         remove_mark(r);
509         (*r).type() = E::member;
510       }
511
512       void unmark_action_1(E*, E* q, E*) {
513         remove_node(q);
514         (*q).type() = E::unmarked;
515       }
516
517       void unmark_action_2(E* p, E* q, E*) {
518         remove_mark(p);
519         (*p).type() = E::singleton;
520         insert_mark(p);
521         remove_node(q);
522         (*q).type() = E::unmarked;
```

```
523       }
524
525       void unmark_action_3(E*, E* q, E* r) {
526         remove_node(q);
527         (*q).type() = E::unmarked;
528         (*r).type() = E::singleton;
529         insert_mark(r);
530       }
531
532       void unmark_action_4(E*, E* q, E* r) {
533         remove_node(q);
534         (*q).type() = E::unmarked;
535         (*r).type() = E::leader;
536         insert_mark(r);
537       }
538
539       void unmark_action_5(E* p, E* q, E* r) {
540         remove_mark(p);
541         (*p).type() = E::singleton;
542         insert_mark(p);
543         remove_node(q);
544         (*q).type() = E::unmarked;
545         (*r).type() = E::singleton;
546         insert_mark(r);
547       }
548
549       void unmark_action_6(E* p, E* q, E* r) {
550         remove_mark(p);
551         (*p).type() = E::singleton;
552         insert_mark(p);
553         remove_node(q);
554         (*q).type() = E::unmarked;
555         (*r).type() = E::leader;
556         insert_mark(r);
557       }
558
559       void unmark_action_7(E*, E* q, E* r) {
560         remove_mark(q);
561         remove_node(q);
562         (*q).type() = E::unmarked;
563         (*r).type() = E::singleton;
564         insert_mark(r);
565       }
566
567       void unmark_action_8(E*, E* q, E* r) {
568         remove_mark(q);
569         remove_node(q);
570         (*q).type() = E::unmarked;
571         (*r).type() = E::leader;
572         insert_mark(r);
573         assert(is_valid());
574       }
575
576       void unmark_action_9(E*, E* q, E*) {
577         remove_mark(q);
578         remove_node(q);
579         (*q).type() = E::unmarked;
580       }
581
582       template <typename H>
583       void cleanparent_transformation(E* q, H& heap_store) {
584          // std::cout << " cleanparent trans " << q  << std::endl;
585         assert(is_marked(q));
586         E* p = (*q).parent();
587         assert((*p).type() == E::unmarked);
588         assert((*p).left() == q);
589         E* r = (*p).right();
590         assert((*r).type() == E::unmarked);
591         unmark(q);
592         (*p).left() = r;
593         (*p).right() = q;
594         E* a = (*q).left();
595         E* c = (*r).left();
596         (*q).left() = c;
```

```
597         (*r).left() = a;
598         if (a != 0) {
599       (*a).parent() = r;
600         }
601         if (c != 0) {
602       (*c).parent() = q;
603         }
604
605         //       mark(q);
606
607         // p = (*q).parent();
608         assert((*p).right() == q);
609         assert(is_valid());
610
611         if (is_marked(p)) {
612       unmark(p);
613       //  unmark(q);
614       typename E::hole_type hole_at_p = (*p).splice_out();
615       q = (*p).split(comparator);
616       E* r = (*p).basic_join(q, comparator);
617       (*r).splice_in(hole_at_p, heap_store);
618       mark(r);
619         }
620         else {
621       //  unmark(q);
622       typename E::hole_type hole_at_p = (*p).splice_out();
623       (void) (*p).split(comparator);
624       E* r = (*p).basic_join(q, comparator);
625       (*r).splice_in(hole_at_p, heap_store);
626       if (q == r) {
627         mark(r);
628       }
629         }
630         assert(is_valid());
631       }
632
633       public:
634
635       /*
636               p                       p
637             /   \                   /   \
638           [q]     r       ->      r     [q]
639           / \     / \             / \   / \
640          a   b   c   d           a   d c   b
641       */
642
643       void cleaning_transformation(E* q) {
644         assert(is_marked(q));
645         E* p = (*q).parent();
646         assert(! is_marked(p));
647         assert((*p).left() == q);
648         E* r = (*p).right();
649         assert(! is_marked(r));
650         unmark(q);
651         (*p).left() = r;
652         (*p).right() = q;
653         E* a = (*q).left();
654         E* c = (*r).left();
655         (*q).left() = c;
656         (*r).left() = a;
657         if (a != 0) {
658       (*a).parent() = r;
659         }
660         if (c != 0) {
661       (*c).parent() = q;
662         }
663         mark(q);
664         assert(is_valid());
665       }
666
667       /*
668               p                       [q]                     [r]
669             /   \                   /   \                   /   \
670           [q]   [r]     ->        p     r       or        p     q
```

```
671         / \   / \           / \   / \           / \   / \
672        a   b c   d         a   c b   d         a   c d   b
673       */
674
675       E* sibling_transformation(E* q) {
676         assert(q != 0);
677         E* p = (*q).parent();
678         assert(! is_marked(p));
679         assert((*p).left() == q);
680         assert(is_marked(q));
681         E* r = (*p).right();
682         assert(is_marked(r));
683         E* a = (*q).left();
684         E* b = (*q).right();
685         E* c = (*r).left();
686         E* d = (*r).right();
687         E* g = (*p).parent();
688         unmark(q);
689         unmark(r);
690         if (comparator((*q).element(), (*r).element())) {
691       (*p).height() -= 1;
692       (*r).height() += 1;
693       (*r).parent() = g;
694       if ((*g).right() == p) {
695         (*g).right() = r;
696       }
697       else {
698         (*g).left() = r;
699       }
700       (*r).left() = p;
701       (*r).right() = q;
702       (*p).parent() = r;
703       (*q).parent() = r;
704       (*p).left() = a;
705       (*p).right() = c;
706       (*q).left() = b;
707       (*q).right() = d;
708       if (a != 0) {
709         (*a).parent() = p;
710         (*c).parent() = p;
711         (*b).parent() = q;
712         (*d).parent() = q;
713       }
714       mark(r);
715       assert(is_valid());
716       return r;
717         }
718         (*p).height() -= 1;
719         (*q).height() += 1;
720         (*q).parent() = g;
721         if ((*g).right() == p) {
722       (*g).right() = q;
723         }
724         else {
725       (*g).left() = q;
726         }
727         (*q).left() = p;
728         (*q).right() = r;
729         (*p).parent() = q;
730         (*r).parent() = q;
731         (*p).left() = a;
732         (*p).right() = c;
733         (*r).left() = b;
734         (*r).right() = d;
735         if (a != 0) {
736       (*a).parent() = p;
737       (*c).parent() = p;
738       (*b).parent() = r;
739       (*d).parent() = r;
740         }
741         mark(q);
742         assert(is_valid());
743         return q;
744       }
```

```
745
746     protected:
747
748       /*
749            (p)                   (p)                   [q]
750           /   \                 /   \                 /   \
751          a     [q]     ->      a     q       or      a     p
752                / \                   / \                   / \
753               b   c                 b   c                 c   b
754       */
755
756       template <typename H>
757       void parent_transformation(E* q, H& heap_store) {
758          // std::cout << " parent trans " << q  << std::endl;
759         assert(is_marked(q));
760         E* p = (*q).parent();
761         assert((*p).right() == q);
762         assert(is_valid());
763         if (is_marked(p)) {
764     unmark(p);
765     unmark(q);
766     typename E::hole_type hole_at_p = (*p).splice_out();
767     q = (*p).split(comparator);
768     E* r = (*p).basic_join(q, comparator);
769     (*r).splice_in(hole_at_p, heap_store);
770     mark(r);
771         }
772         else {
773     unmark(q);
774     typename E::hole_type hole_at_p = (*p).splice_out();
775     (void) (*p).split(comparator);
776     E* r = (*p).basic_join(q, comparator);
777     (*r).splice_in(hole_at_p, heap_store);
778     if (q == r) {
779       mark(r);
780     }
781         }
782         assert(is_valid());
783       }
784
785       /*
786         p          r              p          r       [q]       [s]
787          \          \               \          \       \         \
788          [q]   ,    [s]   ->        r    or    p   ,   s   or    q
789          / \        / \            / \        / \     / \       / \
790         a   b      c   d          a   c      c   a   b   d     d   b
791       */
792
793       template <typename H>
794       void pair_transformation(E* q, E* s, H& heap_store) {
795          // std::cout << " pair trans " << q << "," << s << std::endl;
796         assert(q != s);
797         assert((*q).height() == (*s).height());
798         assert(! (*q).is_root() && ! (*s).is_root());
799         E* p = (*q).parent();
800         assert(q == (*p).right());
801         unmark(q);
802         E* r = (*s).parent();
803         assert(s == (*r).right());
804         unmark(s);
805         typename E::hole_type hole_at_p = (*p).splice_out();
806         typename E::hole_type hole_at_r = (*r).splice_out();
807         (void) (*p).split(comparator);
808         (void) (*r).split(comparator);
809         E* t = (*p).basic_join(r, comparator);
810         E* u = (*q).basic_join(s, comparator);
811         if (p == t) {
812     (*t).splice_in(hole_at_p, heap_store);
813     (*u).splice_in(hole_at_r, heap_store);
814         }
815         else {
816     (*t).splice_in(hole_at_r, heap_store);
817     (*u).splice_in(hole_at_p, heap_store);
818         }
```

```
819         mark(u);
820         assert(is_valid());
821       }
822
823       template <typename H>
824       void singleton_transformation(E* q, E* s, H& heap_store) {
825          // std::cout << " singleton trans " << q << "," << s << std::endl;
826         assert(is_marked(q));
827         assert(is_marked(s));
828         E* p = (*q).parent();
829         if ((*p).right() == q) {
830     if (is_marked(p)) {
831       parent_transformation(q, heap_store);
832       return;
833     }
834     if (! (*p).is_root() && is_marked((*p).left())) {
835       sibling_transformation((*p).left());
836       return;
837     }
838         }
839         else {
840     if (is_marked((*p).right())) {
841       if (is_marked(p)) {
842         parent_transformation((*p).right(), heap_store);
843         return;
844       }
845       sibling_transformation(q);
846       return;
847     }
848     cleaning_transformation(q);
849         }
850         E* r = (*s).parent();
851         if ((*r).right() == s) {
852     if (is_marked(r)) {
853       parent_transformation(s, heap_store);
854       return;
855     }
856     if (! (*r).is_root() && is_marked((*r).left())) {
857       sibling_transformation((*r).left());
858       return;
859     }
860         }
861         else {
862     if (is_marked((*r).right())) {
863       if (is_marked(r)) {
864         parent_transformation((*r).right(), heap_store);
865         return;
866       }
867       sibling_transformation(s);
868       return;
869     }
870     cleaning_transformation(s);
871         }
872         pair_transformation(q, s, heap_store);
873         assert(is_valid());
874       }
875
876       template <typename H>
877       void run_transformation(E* q, H& heap_store) {
878         assert((*q).type() == E::leader);
879         E* r = (*q).left();
880         assert((*r).type() == E::member);
881         E* p = (*q).parent();
882         if ((*p).right() == q) {
883     if (is_marked(p)) {
884       // std::cout << " only parent" << std::endl;
885       parent_transformation(q, heap_store);
886       return;
887     }
888     // std::cout << " first parent" << std::endl;
889     parent_transformation(q, heap_store);
890     if (! is_marked(q)) {
891       return;
892     }
```

```
893     if (is_marked((*p).left())) {
894       sibling_transformation((*p).left());
895       return;
896     }
897     // std::cout << " second parent" << std::endl;
898     parent_transformation(r, heap_store);
899     if (! is_marked(r)) {
900       return;
901     }
902     // std::cout << " third parent" << std::endl;
903     parent_transformation(r, heap_store);
904         }
905         else {
906     if (is_marked((*p).right())) {
907       sibling_transformation(q);
908       return;
909     }
910     // std::cout << " 1st cleaning " << std::endl;
911     cleaning_transformation(q);
912
913
914   #ifdef RANK
915     if (teams.size()>0) {
916       //  std::cout << " ## special case " << std::endl;
917           return;
918     }
919   #endif
920     assert((*r).parent() == (*p).left());
921     if (is_marked((*r).parent() -> right())) {
922       sibling_transformation(r);
923       return;
924     }
925     /// std::cout << " 2nd cleaning " << std::endl;
926     cleanparent_transformation(r, heap_store);
927     //  cleaning_transformation(r);
928     // std::cout << " 1st parent " << std::endl;
929     //  parent_transformation(r, heap_store);
930     if (! is_marked(r)) {
931       return;
932     }
933     //  std::cout << " second sibling" << std::endl;
934     sibling_transformation(r);
935         }
936         assert(is_valid());
937       }
938
939     };
940   }
941
942   #if defined(UNITTEST_LAZY_MARK_STORE)
943   #include "proxy-list-heap-store.h++"
944   #include "fat-weak-heap-node.h++"
945   #include <functional>
946   #include <memory>
947   #include "heap-proxy.h++"
948   #include "multiple-heap-framework.h++"
949
950   template <typename V, typename E, typename A>
951   E* create(V const& v, A& allocator) {
952     E* p = allocator.allocate(1);
953     new (p) E(v, allocator);
954     return p;
955   }
956
957   template <typename V, typename E, typename A>
958   void destroy(E* p, A& allocator) {
959     p->~E();
960     allocator.deallocate(p, 1);
961   }
962
963   template <typename V, typename C, typename A, typename E,
964       typename H, typename M>
965   void test_mark_store() {
966     typedef cphstl::multiple_heap_framework<V, C, A, E, H, M> Q;
```

```
967     Q queue;
968
969     typedef typename A::template rebind<E>::other node_allocator_type;
970     node_allocator_type node_allocator;
971     E* p = create<V, E, node_allocator_type>(V(2), node_allocator);
972     E* q = create<V, E, node_allocator_type>(V(4), node_allocator);
973     E* r = create<V, E, node_allocator_type>(V(1), node_allocator);
974     E* s = create<V, E, node_allocator_type>(V(7), node_allocator);
975     E* t = create<V, E, node_allocator_type>(V(11), node_allocator);
976     E* u = create<V, E, node_allocator_type>(V(3), node_allocator);
977     E* v = create<V, E, node_allocator_type>(V(5), node_allocator);
978     E* w = create<V, E, node_allocator_type>(V(8), node_allocator);
979     E* x = create<V, E, node_allocator_type>(V(9), node_allocator);
980     E* y = create<V, E, node_allocator_type>(V(6), node_allocator);
981     E* z = create<V, E, node_allocator_type>(V(10), node_allocator);
982
983     queue.insert(p);
984     queue.insert(q);
985     queue.insert(r);
986     queue.insert(s);
987     queue.insert(t);
988     queue.insert(u);
989     queue.insert(v);
990     queue.insert(w);
991     queue.insert(x);
992     queue.insert(y);
993     queue.insert(z);
994     assert(queue.size() == 11);
995     assert(queue.top() == t);
996
997     H heap_store;
998     M mark_store;
999
1000    heap_store.inject(t, 3, mark_store);
1001    heap_store.inject(x, 1, mark_store);
1002    heap_store.inject(z, 0, mark_store);
1003
1004    assert(heap_store.is_valid(mark_store));
1005    assert(mark_store.is_valid());
1006
1007    mark_store.mark(u);
1008    assert(heap_store.is_valid(mark_store));
1009    assert(mark_store.is_valid());
1010    mark_store.mark(v);
1011    assert(heap_store.is_valid(mark_store));
1012    assert(mark_store.is_valid());
1013    mark_store.mark(w);
1014    assert(heap_store.is_valid(mark_store));
1015    assert(mark_store.is_valid());
1016    mark_store.mark(r);
1017    assert(heap_store.is_valid(mark_store));
1018    assert(mark_store.is_valid());
1019    mark_store.mark(s);
1020    assert(heap_store.is_valid(mark_store));
1021    assert(mark_store.is_valid());
1022    mark_store.mark(q);
1023    assert(heap_store.is_valid(mark_store));
1024    assert(mark_store.is_valid());
1025    mark_store.mark(p);
1026    assert(heap_store.is_valid(mark_store));
1027    assert(mark_store.is_valid());
1028    mark_store.mark(t);
1029    assert(heap_store.is_valid(mark_store));
1030    assert(mark_store.is_valid());
1031    mark_store.mark(x);
1032    assert(heap_store.is_valid(mark_store));
1033    assert(mark_store.is_valid());
1034    mark_store.mark(y);
1035    assert(heap_store.is_valid(mark_store));
1036    assert(mark_store.is_valid());
1037    mark_store.mark(z);
1038    assert(heap_store.is_valid(mark_store));
1039    assert(mark_store.is_valid());
1040
```

```
1041    for (int i = 0; i != 11; ++i) {
1042      mark_store.reduce(heap_store);
1043      assert(heap_store.is_valid(mark_store));
1044      assert(mark_store.is_valid());
1045    }
1046
1047    mark_store.unmark(z);
1048    assert(heap_store.is_valid(mark_store));
1049    assert(mark_store.is_valid());
1050    mark_store.unmark(y);
1051    assert(heap_store.is_valid(mark_store));
1052    assert(mark_store.is_valid());
1053    mark_store.unmark(x);
1054    assert(heap_store.is_valid(mark_store));
1055    assert(mark_store.is_valid());
1056    mark_store.unmark(t);
1057    assert(heap_store.is_valid(mark_store));
1058    assert(mark_store.is_valid());
1059    mark_store.unmark(p);
1060    assert(heap_store.is_valid(mark_store));
1061    assert(mark_store.is_valid());
1062    mark_store.unmark(q);
1063    assert(heap_store.is_valid(mark_store));
1064    assert(mark_store.is_valid());
1065    mark_store.unmark(s);
1066    assert(heap_store.is_valid(mark_store));
1067    assert(mark_store.is_valid());
1068    mark_store.unmark(r);
1069    assert(heap_store.is_valid(mark_store));
1070    assert(mark_store.is_valid());
1071    mark_store.unmark(w);
1072    assert(heap_store.is_valid(mark_store));
1073    assert(mark_store.is_valid());
1074    mark_store.unmark(v);
1075    assert(heap_store.is_valid(mark_store));
1076    assert(mark_store.is_valid());
1077    mark_store.unmark(u);
1078    assert(heap_store.is_valid(mark_store));
1079    assert(mark_store.is_valid());
1080
1081    std::pair<E*, std::size_t> small = heap_store.eject();
1082    assert(small.first == z);
1083    assert(small.second == 0);
1084
1085    std::pair<E*, std::size_t> medium = heap_store.eject();
1086    assert(medium.first == x);
1087    assert(medium.second == 1);
1088
1089    std::pair<E*, std::size_t> big = heap_store.eject();
1090    assert(big.first == t);
1091    assert(big.second == 3);
1092
1093    /* The above code destroyes owners.
1094    queue.show();
1095    while (queue.size() > 0) {
1096      (void) queue.extract();
1097    }
1098    */
1099
1100    destroy<V, E, node_allocator_type>(p, node_allocator);
1101    destroy<V, E, node_allocator_type>(q, node_allocator);
1102    destroy<V, E, node_allocator_type>(r, node_allocator);
1103    destroy<V, E, node_allocator_type>(s, node_allocator);
1104    destroy<V, E, node_allocator_type>(t, node_allocator);
1105    destroy<V, E, node_allocator_type>(u, node_allocator);
1106    destroy<V, E, node_allocator_type>(v, node_allocator);
1107    destroy<V, E, node_allocator_type>(w, node_allocator);
1108    destroy<V, E, node_allocator_type>(x, node_allocator);
1109    destroy<V, E, node_allocator_type>(y, node_allocator);
1110    destroy<V, E, node_allocator_type>(z, node_allocator);
1111  }
1112
1113  int main(int, char**) {
1114    typedef int V;
```

```
1115    typedef std::less<V> C;
1116    typedef std::allocator<V> A;
1117    typedef cphstl::fat_weak_heap_node<V, A> E;
1118    typedef cphstl::proxy_list_heap_store<C, A, E> H;
1119    typedef cphstl::lazy_mark_store<C, A, E> M;
1120
1121    test_mark_store<V, C, A, E, H, M>();
1122  }
1123
1124  #endif
1125  #endif
```

G.10 multiple-heap-framework.h++

```
1   /*
2     A priority-queue framework can be used to generate a realizator for
3     a meldable priority queue. The efficiency of different operations
4     depends on the policies relied on.
5
6     Author: Jyrki Katajainen Â© 2009, 2010
7   */
8
9   #ifndef _CPHSTL_MULTIPLE_HEAP_FRAMEWORK_
10  #define _CPHSTL_MULTIPLE_HEAP_FRAMEWORK_
11
12  #include "allocator-proxy.h++"
13  #include "blank-mark-store.h++"
14  #include "comparator-proxy.h++"
15  #include <cstddef> // std::size_t and std::ptrdiff_t
16  #include "proxy-list-heap-store.h++"
17  #include <functional> // std::less
18  #include <memory> // std::allocator
19  #include "weak-heap-node.h++"
20
21  namespace cphstl {
22
23    template <
24      typename V,
25      typename C = std::less<V>,
26      typename A = std::allocator<V>,
27      typename E = weak_heap_node<V, A>,
28      typename H = proxy_list_heap_store<C, A, E>,
29      typename M = blank_mark_store<C, A, E>
30    >
31    class multiple_heap_framework {
32    public:
33
34      // types
35
36      typedef V value_type;
37      typedef C comparator_type;
38      typedef A allocator_type;
39      typedef E encapsulator_type;
40      typedef H heap_store_type;
41      typedef M mark_store_type;
42      typedef std::size_t size_type;
43      typedef V& reference;
44      typedef V const& const_reference;
45
46      // structors
47
48      explicit multiple_heap_framework(C const& = C(), A const& = A());
49      ~multiple_heap_framework();
50
51      // iterators
52
53      E* begin() const;
54      E* end() const;
55
56      // accessors
57
58      A get_allocator() const;
59      C get_comparator() const;
```

```
60      size_type size() const;
61      size_type max_size() const;
62      E* top() const;
63
64      // modifiers
65
66      E* insert(E*);
67      E* extract();
68      void extract(E*);
69      void increase(E*, V const&);
70      void meld(multiple_heap_framework&);
71      void swap(multiple_heap_framework&);
72
73  #ifdef DEBUG
74
75      void show() {
76        heap_store.show();
77      }
78
79  #endif
80
81    protected:
82
83      comparator_proxy<C> comparator;
84      allocator_proxy<A> allocator;
85      H heap_store;
86      M mark_store;
87      E* top_;
88      size_type size_;
89
90    private:
91
92      multiple_heap_framework(multiple_heap_framework const&);
93      multiple_heap_framework& operator = (multiple_heap_framework const&);
94    };
95
96  }
97
98  #include "multiple-heap-framework.i++"
99  #endif
```

G.11 multiple-heap-framework.i++

```
1   /*
2     Implementation of the multiple-heap priority-queue framework.
3
4     Author: Jyrki Katajainen Â© 2009, 2010
5   */
6
7   #include <algorithm> // std::swap
8   #include "assert.h++"
9   #include <climits> // LONG_MAX
10  #include <iostream>
11  #include <list>
12  #include <vector>
13  #include <utility> // std::pair
14
15  namespace cphstl {
16
17    template <typename V, typename C, typename A, typename E,
18        typename H, typename M>
19    multiple_heap_framework<V, C, A, E, H, M>::multiple_heap_framework(
20      C const& c, A const& a)
21      : comparator(c), allocator(a), heap_store(c, a), mark_store(c), top_(0),
22        size_(0) {
23    }
24
25    template <typename V, typename C, typename A, typename E,
26        typename H, typename M>
27    multiple_heap_framework<V, C, A, E, H, M>::~multiple_heap_framework() {
28      // Precondition: The data structure contains no elements.
29    }
30
```

```
31    template <typename V, typename C, typename A, typename E,
32        typename H, typename M>
33    E*
34    multiple_heap_framework<V, C, A, E, H, M>::begin() const {
35      if (size_ == 0) {
36        return (E*) 0;
37      }
38      return (*heap_store.begin()).root();
39    }
40
41    template <typename V, typename C, typename A, typename E,
42        typename H, typename M>
43    E*
44    multiple_heap_framework<V, C, A, E, H, M>::end() const {
45      return (E*) 0;
46    }
47
48    template <typename V, typename C, typename A, typename E,
49        typename H, typename M>
50    A
51    multiple_heap_framework<V, C, A, E, H, M>::get_allocator() const {
52      return allocator.subject();
53    }
54
55    template <typename V, typename C, typename A, typename E,
56        typename H, typename M>
57    C
58    multiple_heap_framework<V, C, A, E, H, M>::get_comparator() const {
59      return comparator.subject();
60    }
61
62    template <typename V, typename C, typename A, typename E,
63        typename H, typename M>
64    typename multiple_heap_framework<V, C, A, E, H, M>::size_type
65    multiple_heap_framework<V, C, A, E, H, M>::size() const {
66      return size_;
67    }
68
69    template <typename V, typename C, typename A, typename E,
70        typename H, typename M>
71    typename multiple_heap_framework<V, C, A, E, H, M>::size_type
72    multiple_heap_framework<V, C, A, E, H, M>::max_size() const {
73      // Assume that n <= LONG_MAX.
74      typename std::vector<int, A>::allocator_type a;
75      size_type n = LONG_MAX;
76      size_type available_memory = a.max_size() * sizeof(int); // in bytes
77      size_type space_available_for_encapsulators = available_memory -
78        heap_store.footprint(n) - mark_store.footprint(n);
79      return space_available_for_encapsulators / E::footprint();
80    }
81
82    template <typename V, typename C, typename A, typename E,
83        typename H, typename M>
84    E*
85    multiple_heap_framework<V, C, A, E, H, M>::top() const {
86      return top_;
87    }
88
89    template <typename V, typename C, typename A, typename E,
90        typename H, typename M>
91    E*
92    multiple_heap_framework<V, C, A, E, H, M>::insert(E* p) {
93
94      heap_store.inject(p, 0, mark_store);
95      if (top_ == 0 || comparator((*top_).element(), (*p).element())) {
96        top_ = p;
97      }
98      ++size_;
99
100   #ifdef DEBUG
101
102     assert(heap_store.is_valid(mark_store));
103     assert(mark_store.is_valid());
104
```

```
105 #endif
106
107     return p;
108   }
109
110   template <typename V, typename C, typename A, typename E,
111       typename H, typename M>
112   void
113   multiple_heap_framework<V, C, A, E, H, M>::increase(E* p, V const& v) {
114     assert(! comparator(v, (*p).element()));
115     (*p).element() = v;
116     mark_store.mark(p);
117     mark_store.reduce(heap_store);
118     if (comparator((*top_).element(), (*p).element())) {
119       top_ = p;
120     }
121
122 #ifdef DEBUG
123
124     assert(heap_store.is_valid(mark_store));
125     assert(mark_store.is_valid());
126
127 #endif
128
129   }
130
131   template <typename V, typename C, typename A, typename E,
132       typename H, typename M>
133   E*
134   multiple_heap_framework<V, C, A, E, H, M>::extract() {
135     assert(size() != 0);
136     std::pair<E*, size_type> pair  = heap_store.eject();
137     assert(heap_store.is_valid(mark_store));
138     E* q = pair.first;
139     size_type h = pair.second;
140     if (h > 0) {
141       E* c = (*q).release_root();
142       heap_store.concatenate(c, h - 1, mark_store);
143     }
144     mark_store.reduce(heap_store);
145     assert(heap_store.is_valid(mark_store));
146     --size_;
147     if (top_ != q) {
148       return q;
149     }
150     if (top_ == q && heap_store.begin() == 0) {
151       top_ = 0;
152       return q;
153     }
154     pair  = heap_store.eject();
155     assert(heap_store.is_valid(mark_store));
156     E* r = pair.first;
157     h = pair.second;
158     (*q).swap_roots(r);
159     assert(heap_store.is_valid(mark_store));
160     heap_store.inject(q, h, mark_store);
161
162 #ifdef DEBUG
163
164     assert(heap_store.is_valid(mark_store));
165     assert(mark_store.is_valid());
166
167 #endif
168
169     return r;
170   }
171
172 #ifdef FIRST_VERSION
173
174   template <typename V, typename C, typename A, typename E,
175       typename H, typename M>
176   void
177   multiple_heap_framework<V, C, A, E, H, M>::extract(E* r) {
178     assert(r != 0);
```

```
179       E* b = extract();
180       assert(b != 0);
181       if (b != r) {
182         E* p = (*r).distinguished_ancestor();
183         while (p != 0) {
184     (*r).swap_nodes(p);
185     p = (*r).distinguished_ancestor();
186         }
187         std::list<E*, A> stack;
188         while (! (*r).is_leaf()) {
189     E* s = (*r).split(comparator);
190     stack.push_front(s);
191         }
192         while (! stack.empty()) {
193     E* s = stack.front();
194     b = (*b).join(s, comparator, mark_store);
195     stack.pop_front();
196         }
197         typedef typename H::heap_proxy_type heap_proxy_type;
198         heap_store.replace((heap_proxy_type*) (*r).owner(), b);
199       }
200       if (top_ == r) {
201         top_ = heap_store.find_top();
202         E* t = mark_store.find_top();
203         if (t != 0 && comparator((*top_).element(), (*t).element())) {
204     top_ = t;
205         }
206       }
207   #ifdef DEBUG
208
209       assert(heap_store.is_valid(mark_store));
210       assert(mark_store.is_valid());
211
212   #endif
213
214     }
215
216   #else
217
218     template <typename V, typename C, typename A, typename E,
219         typename H, typename M>
220     void
221     multiple_heap_framework<V, C, A, E, H, M>::extract(E* p) {
222       assert(p != 0);
223       E* replacement = extract();
224       assert(heap_store.is_valid(mark_store));
225       assert(replacement != 0);
226       if (p == replacement) {
227         return;
228       }
229       mark_store.unmark(p);
230       E* q = p;
231       typename E::hole_type hole_at_p = (*p).splice_out();
232       E* s = (*p).distinguished_descendant(comparator);
233       while (s != 0) {
234         mark_store.unmark(s);
235         q = s;
236         s = (*s).distinguished_descendant(comparator);
237       }
238
239       E* r = replacement;
240       while (q != p) {
241         E* s = q;
242         q = (*q).parent();
243         r = (*r).fast_join(s, replacement, comparator, mark_store);
244       }
245       (*r).splice_in(hole_at_p, heap_store);
246       mark_store.mark(r);
247       mark_store.reduce(heap_store);
248       if (p == top_) {
249         top_ = heap_store.find_top();
250         E* t = mark_store.find_top();
251         if (t != 0 && comparator((*top_).element(), (*t).element())) {
252     top_ = t;
```

```
253         }
254       }
255
256   #ifdef DEBUG
257
258       assert(heap_store.is_valid(mark_store));
259       assert(mark_store.is_valid());
260
261   #endif
262
263     }
264
265   #endif
266
267     template <typename V, typename C, typename A, typename E,
268         typename H, typename M>
269     void
270     multiple_heap_framework<V, C, A, E, H, M>::meld(
271       multiple_heap_framework& other) {
272       // Precondition: The allocators and comparators must be compatible.
273       if (size_ < other.size_) {
274         swap(other);
275       }
276       heap_store.meld(other.heap_store, mark_store);
277       mark_store.meld(other.mark_store, heap_store);
278       size_ += other.size_;
279       other.size_ = 0;
280       if (other.top_ != 0 &&
281     comparator((*top_).element(), (*other.top_).element())) {
282         top_ = other.top_;
283       }
284       other.top_ = 0;
285
286
287   #ifdef DEBUG
288
289       assert(heap_store.is_valid(mark_store));
290       assert(other.heap_store.is_valid(other.mark_store));
291
292   #endif
293
294     }
295
296     template <typename V, typename C, typename A, typename E,
297         typename H, typename M>
298     void
299     multiple_heap_framework<V, C, A, E, H, M>::swap(
300       multiple_heap_framework& other) {
301       // Precondition: The allocators and comparators are compatible.
302       heap_store.swap(other.heap_store);
303       mark_store.swap(other.mark_store);
304       std::swap(top_, other.top_);
305       std::swap(size_, other.size_);
306     }
307   }
308
309   #if defined(UNITTEST_MULTIPLE_HEAP_FRAMEWORK)
310   #include "blank-mark-store.h++"
311   #include "proxy-list-heap-store.h++"
312   #include "fat-weak-heap-node.h++"
313   #include <functional>
314   #include "lazy-mark-store.h++"
315   #include <memory>
316   #include "multiple-heap-framework.h++"
317   #include "pennant-node.h++"
318   #include "heap-proxy.h++"
319   #include "root-list-heap-store.h++"
320   #include "weak-heap-node.h++"
321
322   #include <algorithm>
323   #include <numeric>
324   #include <vector>
325
326   using namespace cphstl;
```

```
327
328 template <typename Q>
329 struct pqfw_test {
330   typedef typename Q::value_type V;
331   typedef typename Q::comparator_type C;
332   typedef typename Q::allocator_type A;
333   typedef typename Q::encapsulator_type E;
334   typedef typename Q::heap_store_type H;
335   typedef typename Q::mark_store_type M;
336
337   template <typename V, typename E, typename A>
338   static E* create(V const& v, A& allocator) {
339     E* p = allocator.allocate(1);
340     new (p) E(v, allocator);
341     return p;
342   }
343
344   template <typename V, typename E, typename A>
345   static void destroy(E* p, A& allocator) {
346     p->~E();
347     allocator.deallocate(p, 1);
348   }
349
350   static void small_test_multiple_heap_framework() {
351     Q w;
352
353     typedef typename A::template rebind<E>::other node_allocator_type;
354     node_allocator_type node_allocator;
355     E* p = create<V, E, node_allocator_type>(V(2), node_allocator);
356     E* q = create<V, E, node_allocator_type>(V(4), node_allocator);
357     E* r = create<V, E, node_allocator_type>(V(1), node_allocator);
358
359     std::cout << "insert\n";
360     w.insert(p);
361     assert(w.top() == p);
362     w.show();
363     assert(w.size() == 1);
364     std::cout << "insert␣" << (*q).element() << "\n";
365     w.insert(q);
366     assert(w.top() == q);
367     w.show();
368     assert(w.size() == 2);
369     std::cout << "extract\n";
370     E* s = w.extract();
371     w.show();
372     assert(w.size() == 1);
373     std::cout << "insert\n";
374     w.insert(r);
375     assert(w.top() != s && w.top() != r);
376     w.show();
377     assert(w.size() == 2);
378     w.extract();
379     w.extract();
380     assert(w.size() == 0);
381
382     destroy<V, E, node_allocator_type>(p, node_allocator);
383     destroy<V, E, node_allocator_type>(q, node_allocator);
384     destroy<V, E, node_allocator_type>(r, node_allocator);
385   }
386
387   static void big_test_multiple_heap_framework() {
388     Q h;
389     assert(h.size() == 0);
390     assert(h.begin() == h.end());
391     typename Q::allocator_type heap_a = h.get_allocator();
392     typename Q::comparator_type heap_c = h.get_comparator();
393     assert(h.size() <= h.max_size());
394     assert(h.top() == h.end());
395     V a[] = {8, 10, 13, 1, 6, 5, 3, 7, 11, 9};
396     unsigned int const n = sizeof(a) / sizeof(V);
397
398     typedef typename A::template rebind<E>::other node_allocator_type;
399     node_allocator_type node_allocator;
400     std::vector<E*, node_allocator_type> node;
```

```
401       for (unsigned int j = 0; j < n; ++j) {
402         E* p = create<V, E, node_allocator_type>(a[j], node_allocator);
403         node.push_back(p);
404       }
405       h.insert(node[0]);
406       assert(h.top() == h.begin());
407       E* t = h.top();
408       assert((*t).element() == V(8));
409       for (unsigned int j = 1; j < n; ++j) {
410         h.insert(node[j]);
411         E* r = h.top();
412         assert((*r).element() == *(std::max_element(&a[0], &a[j + 1])));
413       }
414       assert(h.size() == n);
415
416       V sum = V(0);
417       while (h.size() != 0) {
418         E* p =    h.extract();
419         sum += (*p).element();
420       }
421       assert(sum == std::accumulate(&a[0], &a[n], V(0)));
422
423       Q g;
424       for (unsigned int j = 0; j < n; ++j) {
425         g.insert(node[j]);
426       }
427       assert(g.size() == n);
428
429       std::sort(&a[0], &a[n]);
430       V another_sum = V(0);
431       while (g.size() != 0) {
432         E* p = g.top();
433         assert((*p).element() == a[g.size() - 1]);
434         another_sum += (*p).element();
435         g.extract(p);
436       }
437       assert(sum == another_sum);
438
439       for (unsigned int j = 0; j < n; ++j) {
440         destroy<V, E, node_allocator_type>(node[j], node_allocator);
441       }
442     }
443   };
444
445   template <typename V, typename C, typename A, typename E,
446     typename H, typename M>
447   void small_test_multiple_heap_framework() {
448       pqfw_test<multiple_heap_framework<V, C, A, E, H, M> >::small_test_multiple_heap_framework
              ();
449   }
450
451   template <typename V, typename C, typename A, typename E,
452     typename H, typename M>
453   void big_test_multiple_heap_framework() {
454       pqfw_test<multiple_heap_framework<V, C, A, E, H, M> >::big_test_multiple_heap_framework();
455   }
456
457   int main(int, char**) {
458     typedef int V;
459     typedef std::less<V> C;
460     typedef std::allocator<V> A;
461
462     std::cout << "weak queues...\n";
463     typedef cphstl::weak_heap_node<V, A> E1; //<--
464     typedef cphstl::proxy_list_heap_store<C, A, E1> H1;
465     typedef cphstl::blank_mark_store<C, A, E1> M1;
466     small_test_multiple_heap_framework<V, C, A, E1, H1, M1>();
467     big_test_multiple_heap_framework<V, C, A, E1, H1, M1>();
468
469     std::cout << "weak queues with fat nodes...\n";
470     typedef cphstl::fat_weak_heap_node<V, A> E2; //<--
471     typedef cphstl::proxy_list_heap_store<C, A, E2> H2;
472     typedef cphstl::blank_mark_store<C, A, E2> M2; //<--
473     small_test_multiple_heap_framework<V, C, A, E2, H2, M2>();
```

```
474     big_test_multiple_heap_framework<V, C, A, E2, H2, M2>();
475
476     std::cout << "pennant queues...\n";
477     typedef cphstl::pennant_node<V, A> E3; //<--
478     typedef cphstl::proxy_list_heap_store<C, A, E3> H3;
479     typedef cphstl::blank_mark_store<C, A, E3> M3;
480     small_test_multiple_heap_framework<V, C, A, E3, H3, M3>();
481     big_test_multiple_heap_framework<V, C, A, E3, H3, M3>();
482
483     std::cout << "run-relaxed weak queues...\n";
484     typedef cphstl::fat_weak_heap_node<V, A> E4; //<--
485     typedef cphstl::proxy_list_heap_store<C, A, E4> H4;
486     typedef cphstl::lazy_mark_store<C, A, E4> M4; //<--
487     small_test_multiple_heap_framework<V, C, A, E4, H4, M4>();
488     big_test_multiple_heap_framework<V, C, A, E4, H4, M4>();
489
490     /*
491     typedef cphstl::weak_heap_node<V, A> E5; //<--
492     typedef cphstl::root_list_heap_store<C, A, E5> H5;
493     typedef cphstl::lazy_mark_store<C, A, E4> M5; //<--
494     small_test_multiple_heap_framework<V, C, A, E5, H5, M5>();
495     big_test_multiple_heap_framework<V, C, A, E5, H5, M5>();
496
497     */
498     return 0;
499 }
500
501 #endif
```

G.12 pennant-node.h++

```
1  /*
2    A pennant node
3
4    Author: Jyrki Katajainen Â© 2009
5  */
6
7  #ifndef _CPHSTL_PENNANT_NODE_
8  #define _CPHSTL_PENNANT_NODE_
9
10 #include <algorithm> // std::swap
11 #include "heap-node.h++"
12
13 namespace cphstl {
14
15   template <typename V, typename A>
16   class pennant_node
17     : public heap_node<V, A, pennant_node<V, A> > {
18
19   private:
20
21     pennant_node();
22     pennant_node(pennant_node const&);
23     pennant_node& operator = (pennant_node const&);
24
25   public:
26
27     typedef V value_type;
28     typedef A allocator_type;
29     typedef pennant_node<V, A> N;
30
31     struct hole_type {
32       N* parent;
33       N* current;
34       union {
35   N* left;
36   void* owner;
37       };
38       N* right;
39
40       hole_type(N* p)
41         : parent((*p).parent()), current(p), right((*p).right()) {
42   if (parent != 0) {
```

```
      left = (*p).left();
    }
    else {
      owner = (*p).owner();
    }
        }
      };

      pennant_node(V const& v, A const& a)
        : heap_node<V, A, N>(v, a) {
      }

      template <typename C, typename M>
      N* join(N* q, C const& comparator, M&) {
        N* p = this;
        assert((*p).is_root());
        assert((*q).is_root());
        if (comparator((*p).element(), (*q).element())) {
    N* c = (*q).right();
    if (c != 0) {
      (*c).parent() = p;
    }
    (*p).left() = c;
    (*q).right() = p;
    (*p).parent() = q;
    sift_down(p, comparator);
    return q;
        }
        else {
    N* c = (*p).right();
    if (c != 0) {
      (*c).parent() = q;
    }
    (*q).left() = c;
    (*p).right() = q;
    (*q).parent() = p;
    sift_down(q, comparator);
    return p;
        }
      }

      template <typename C, typename M>
      N* fast_join(N* q, N* replacement, C const& comparator, M&) {
        N* p = this;
        if (p != replacement || comparator((*p).element(), (*q).element())) {
    N* d = (*q).right();
    (*p).parent() = q;
    (*p).left() = d;
    if (d != 0) {
      (*d).parent() = p;
    }
    (*q).right() = p;
    return q;
        }
        N* c = (*p).right();
        (*q).parent() = p;
        (*q).left() = c;
        if (c != 0) {
    (*c).parent() = q;
        }
        (*p).right() = q;
        return p;
      }

      template <typename C>
      N* split(C const&) {
        N* p = this;
        assert(p != 0);
        assert((*p).right() != 0);
        N* q = (*p).right();
        if ((*q).right() == 0) {
    (*p).right() = 0;
    (*q).parent() = 0;
    return q;
```

```
117         }
118         N* r = (*q).left();
119         (*p).right() = r;
120         if (r != 0) {
121       (*r).parent() = p;
122         }
123         (*q).parent() = 0;
124         (*q).left() = 0;
125         return q;
126       }
127
128       template <typename C>
129       N* distinguished_descendant(C const& comparator) {
130         assert(this != 0);
131         if ((*this).is_leaf()) {
132       return 0;
133         }
134         if ((*this).is_root()) {
135       return (*this).right();
136         }
137         N* q = (*this).left();
138         N* r = (*this).right();
139         if (comparator((*q).element(), (*r).element())) {
140       (*this).left() = r;
141       (*this).right() = q;
142       return r;
143         }
144         return q;
145       }
146
147       N* distinguished_ancestor() const {
148         assert(this != 0);
149         return (*this).parent();
150       }
151
152       N* promote(N* p) {
153         N* q = this;
154         assert(p == (*q).distinguished_ancestor());
155         if ((*p).right() == q) {
156       N* a = (*p).parent();
157       N* b = (*p).left();
158       N* f = (*q).left();
159       N* g = (*q).right();
160       (*p).parent() = q;
161       (*p).left() = f;
162       (*p).right() = g;
163       (*q).parent() = a;
164       (*q).left() = b;
165       (*q).right() = p;
166       if (a != 0) {
167         if ((*a).right() == p) {
168           (*a).right() = q;
169         }
170         else {
171           if (! (*p).is_root()) {
172             (*a).left() = q;
173           }
174         }
175       }
176       if (b != 0 && (! (*p).is_root())) {
177         (*b).parent() = q;
178       }
179       if (f != 0) {
180         (*f).parent() = p;
181       }
182       if (g != 0) {
183         (*g).parent() = p;
184       }
185         }
186         else {
187       N* a = (*p).parent();
188       N* c = (*p).right();
189       N* f = (*q).left();
190       N* g = (*q).right();
```

```
191     (*p).parent() = q;
192     (*p).left() = f;
193     (*p).right() = g;
194     (*q).parent() = a;
195     (*q).left() = p;
196     (*q).right() = c;
197     if (a != 0) {
198       if ((*a).right() == p) {
199         (*a).right() = q;
200       }
201       else {
202         if (! (*p).is_root()) {
203           (*a).left() = q;
204         }
205       }
206     }
207     if (c != 0) {
208       (*c).parent() = q;
209     }
210     if (f != 0) {
211       (*f).parent() = p;
212     }
213     if (g != 0) {
214       (*g).parent() = p;
215     }
216         }
217         return q;
218       }
219
220       hole_type splice_out() {
221         assert(this != 0);
222         hole_type hole(this);
223         return hole;
224       }
225
226       template <typename H>
227       void splice_in(hole_type& hole, H& heap_store) {
228         N* here = hole.current;
229         assert(here != 0);
230         N* above = hole.parent;
231         (*this).parent() = above;
232         if (above == 0) {
233     (*this).owner() = hole.owner;
234     heap_store.replace(this);
235     return;
236         }
237         if ((*above).left() == here) {
238     (*above).left() = this;
239         }
240         if ((*above).right() == here) {
241     (*above).right() = this;
242         }
243         if (this == hole.left || (*this).right() == hole.left) {
244     (*this).left() = hole.right;
245     if (hole.right != 0) {
246       (*hole.right).parent() = this;
247     }
248     return;
249         }
250         if (this == hole.right || (*this).right() == hole.right) {
251     (*this).left() = hole.left;
252     if (hole.left != 0) {
253       (*hole.left).parent() = this;
254     }
255     return;
256         }
257         assert(false);
258       }
259
260   #ifdef DEBUG
261
262       template <typename C, typename M>
263       bool is_valid(C const& comparator, M const& mark_store) const {
264         N const* t = (*this).down_cast(this);
```

```
265         bool valid = true;
266         if ((*t).parent() != 0) {
267       valid &= t -> parent() -> left() == t ||
268         t -> parent() -> right() == t;
269       if (! valid) std::cout << "error: parent\n";
270         }
271         if ((*t).left() != 0) {
272       valid &= t -> left() -> parent() == t;
273       if (! valid) std::cout << "error: left\n";
274         }
275         if ((*t).right() != 0) {
276       valid &= t -> right() -> parent() == t;
277       if (! valid) std::cout << "error: right\n";
278         }
279         if (! (*t).is_root() && ! mark_store.is_marked(t)) {
280       valid &= ! comparator((*t).parent() -> element(), (*t).element());
281       if (! valid) {
282         std::cout << "error: heap order\n";
283         (*t).root() -> show_tree();
284       }
285         }
286         return valid;
287       }
288
289   #endif
290
291     protected:
292
293       template <typename C>
294       void sift_down(N* p, C const& comparator) {
295         assert(p != 0);
296         while (! (*p).is_leaf()) {
297       N* q = (*p).left();
298       N* r = (*p).right();
299       if (comparator((*q).element(),(*r).element())) {
300         if (comparator((*p).element(), (*r).element())) {
301           p = (*r).promote(p);
302           p = (*p).right();
303         }
304         else break;
305       }
306       else {
307         if (comparator((*p).element(), (*q).element())) {
308           p = (*q).promote(p);
309           p = (*p).left();
310         }
311         else break;
312       }
313         }
314       }
315     };
316   }
317
318   #endif
```

G.13 proxy-array-heap-store.h++

```
1   /*
2     A proxy-array heap store; the redundant binary system is used for
3     keeping track of the perfect components.
4
5     Author: Jyrki Katajainen Â© 2010
6   */
7
8   #ifndef _CPHSTL_PROXY_ARRAY_HEAP_STORE_
9   #define _CPHSTL_PROXY_ARRAY_HEAP_STORE_
10
11  #include <algorithm> // std::swap
12  #include "assert.h++"
13  #include "bit-store.h++"
14  #include <climits> // CHAR_BIT
15  #include "comparator-proxy.h++"
16  #include <cstddef> // std::size_t
```

```
17 #include <iostream>
18 #include <memory> // std::allocator
19 #include <utility> // std::pair
20 #include <vector>
21
22 namespace cphstl {
23
24   template<
25     typename C,
26     typename A,
27     typename E,
28     typename W = unsigned long
29   >
30   class proxy_array_heap_store {
31   public:
32
33     typedef C comparator_type;
34     typedef E encapsulator_type;
35     typedef A allocator_type;
36     typedef W word_type;
37     typedef std::size_t size_type;
38     typedef proxy_array_heap_store<C, A, E, W> self_type;
39
40     enum {word_size = CHAR_BIT * sizeof(W)};
41
42     class heap_proxy {
43     public:
44
45       E* root_;
46       self_type* store;
47
48       heap_proxy(E* r = 0, self_type* s = 0)
49   : root_(r), store(s) {
50       }
51
52       E* root() const {
53   return root_;
54       }
55
56       E*& root() {
57   return root_;
58       }
59
60       heap_proxy* successor() const {
61   return (*store).next((heap_proxy*) this);
62       }
63     };
64
65     typedef heap_proxy heap_proxy_type;
66
67   protected:
68
69     comparator_proxy<C> comparator;
70     cphstl::bit_store<W> high;
71     cphstl::bit_store<W> low;
72     std::vector<std::pair<heap_proxy, heap_proxy>, A> proxies; // !: ab, A was missing.
73
74   public:
75
76     proxy_array_heap_store(C const& c = C(), A const& a = A())
77       : comparator(c), high(), low(), proxies(a) {
78       typedef std::pair<heap_proxy, heap_proxy> pair_type;
79       pair_type p(heap_proxy((E*) 0, (self_type*) this), heap_proxy((E*) 0, (self_type*) this
          ));
80       proxies.resize(word_size, p);
81     }
82
83     ~proxy_array_heap_store() {
84     }
85
86     size_type footprint(size_type) const {
87       return sizeof(proxy_array_heap_store) + 2 * sizeof(E*) * word_size
88   + 2 * sizeof(self_type*) * word_size;
89     }
```

```
90
91        heap_proxy* begin() const {
92          if (low.get(0)) {
93      return (heap_proxy*) &proxies[0].first;
94          }
95          return next((heap_proxy*) &proxies[0].first);
96        }
97
98        heap_proxy* begin() {
99          if (low.get(0)) {
100     return &proxies[0].first;
101         }
102         return next(&proxies[0].first);
103       }
104
105       heap_proxy* next(heap_proxy* p) const {
106         size_type i = p - &proxies[0].first;
107         if (p == &proxies[i].first) {
108     if (high.get(i)) {
109       return (heap_proxy*) &proxies[i].second;
110     }
111         }
112         W mask = (1 << (i + 1)) - 1;
113         mask = ~mask;
114         W rest = size_type(low) & mask;
115         bit_store<W> word(rest);
116         if (word.size() == 0) {
117     return 0;
118         }
119         size_type j = word.least_significant_one();
120         return (heap_proxy*) &proxies[j].first;
121       }
122
123       E* find_top() const {
124         if (low.size() == 0) {
125     return 0;
126         }
127         bit_store<W> down = low;
128         bit_store<W> up = high;
129         size_type h = down.least_significant_one();
130         E* top = proxies[h].first.root();
131         if (up.get(h)) {
132     E* p = proxies[h].second.root();
133     if (comparator((*top).element(), (*p).element())) {
134       top = p;
135     }
136         }
137         down.unset(h);
138         up.unset(h);
139         while (down.size() != 0) {
140     h = down.least_significant_one();
141     E* p = proxies[h].first.root();
142     if (comparator((*top).element(), (*p).element())) {
143       top = p;
144     }
145     if (up.get(h)) {
146       p = proxies[h].second.root();
147       if (comparator((*top).element(), (*p).element())) {
148         top = p;
149       }
150     }
151     down.unset(h);
152     up.unset(h);
153         }
154         return top;
155       }
156
157       template <typename M>
158       void inject(E* p, size_type h, M& mark_store) {
159         assert(p != 0);
160         insert(p, h);
161         fix(mark_store);
162       }
163
```

```
164       std::pair<E*, size_type> eject() {
165         // Warning: top_ can point to the nodes ejected
166         if (low.size() == 0) {
167       return std::make_pair((E*) 0, (size_type) 0);
168         }
169         size_type h = low.least_significant_one();
170         if (high.get(h)) {
171       E* r = proxies[h].second.root();
172       high.unset(h);
173       return std::make_pair(r, h);
174         }
175         E* r = proxies[h].first.root();
176         low.unset(h);
177         return std::make_pair(r, h);
178       }
179
180       template <typename M>
181       void concatenate(E* q, size_type h, M& mark_store) {
182         while (q != 0) {
183       mark_store.unmark(q);
184       E* s = (*q).release_subheap();
185       inject(q, h, mark_store);
186       h = h - 1;
187       q = s;
188         }
189       }
190
191       void replace(E* new_root) {
192         heap_proxy_type* proxy = (heap_proxy_type*) (*new_root).owner();
193         (*proxy).root() = new_root;
194         (*new_root).owner() = (void *) proxy;
195       }
196
197       template <typename M>
198       void meld(proxy_array_heap_store& other, M& mark_store) {
199         // Precondition: The comparators must be compatible.
200         if (low.size() == 0) {
201       swap(other);
202       return;
203         }
204         if (other.low.size() == 0) {
205       return;
206         }
207         size_type h_1 = low.most_significant_one();
208         size_type h_2 = other.low.most_significant_one();
209         if (h_1 < h_2) {
210       swap(other);
211       std::swap(h_1, h_2);
212         }
213         bit_store<W> high_1 = high;
214         bit_store<W> low_1 = low;
215         bit_store<W> high_2 = other.high;
216         bit_store<W> low_2 = other.low;
217        W mask = (1 << (h_2 + 1)) - 1;
218         mask = ~mask;
219         (*this).low = bit_store<W>(mask & (W) low);
220         (*this).high = bit_store<W>(mask & (W) high);
221         other.low = bit_store<W>();
222         other.high = bit_store<W>();
223         size_type h = h_2;
224         while (true) {
225       W branch = 8 * high_1.get(h) + 4 * low_1.get(h) + 2 * high_2.get(h)
226         + low_2.get(h);
227       E* tmp;
228       switch (branch) {
229         case 0 /* 0000 */:
230           break;
231         case 1 /* 0001 */:
232           insert(other.proxies[h].first.root(), h);
233           fix(mark_store);
234           break;
235         case 2 /* 0010 */:
236           assert(false);
237           break;
```

```
238       case 3 /* 0011 */:
239         insert(other.proxies[h].first.root(), h);
240         fix(mark_store);
241         insert(other.proxies[h].second.root(), h);
242         fix(mark_store);
243         break;
244       case 4 /* 0100 */:
245         insert(proxies[h].first.root(), h);
246         fix(mark_store);
247         break;
248       case 5 /* 0101 */:
249         insert(proxies[h].first.root(), h);
250         fix(mark_store);
251         insert(other.proxies[h].first.root(), h);
252         fix(mark_store);
253         break;
254       case 6 /* 0110 */:
255         assert(false);
256         break;
257       case 7 /* 0111 */:
258         insert(proxies[h].first.root(), h);
259         fix(mark_store);
260         insert(other.proxies[h].first.root(), h);
261         fix(mark_store);
262         insert(other.proxies[h].second.root(), h);
263         fix(mark_store);
264         break;
265       case 8 /* 1000 */:
266         assert(false);
267         break;
268       case 9 /* 1001 */:
269         assert(false);
270         break;
271       case 10 /* 1010 */:
272         assert(false);
273         break;
274       case 11 /* 1011 */:
275         assert(false);
276         break;
277       case 12 /* 1100 */:
278         tmp = proxies[h].second.root();
279         insert(proxies[h].first.root(), h);
280         fix(mark_store);
281         insert(tmp, h);
282         fix(mark_store);
283         break;
284       case 13 /* 1101 */:
285         tmp = proxies[h].second.root();
286         insert(proxies[h].first.root(), h);
287         fix(mark_store);
288         insert(tmp, h);
289         fix(mark_store);
290         insert(other.proxies[h].first.root(), h);
291         fix(mark_store);
292         break;
293       case 14 /* 1110 */:
294         assert(false);
295         break;
296       case 15 /* 1111 */:
297         tmp = proxies[h].second.root();
298         insert(proxies[h].first.root(), h);
299         fix(mark_store);
300         insert(tmp, h);
301         fix(mark_store);
302         insert(other.proxies[h].first.root(), h);
303         fix(mark_store);
304         insert(other.proxies[h].second.root(), h);
305         fix(mark_store);
306         break;
307       default:
308         assert(false);
309       }
310     if (h == 0) {
311       break;
```

```
312       }
313       --h;
314           }
315         }
316
317         void swap(proxy_array_heap_store& other) {
318           std::swap(comparator, other.comparator);
319           std::swap(proxies, other.proxies);
320           std::swap(high, other.high);
321           std::swap(low, other.low);
322         }
323
324     #ifdef DEBUG
325
326         void show() {
327           std::cout << "␣==␣==␣==␣␣=␣\n";
328           cphstl::bit_store<word_type> high_bits = high;
329           std::cout << "high:␣";
330           for (size_type i = 0; i < word_size; ++i) {
331       std::cout << high.get(i);
332           }
333           std::cout << "\n";
334
335           cphstl::bit_store<word_type> low_bits = low;
336           std::cout << "low:␣";
337           for (size_type i = 0; i < word_size; ++i) {
338       std::cout << low.get(i);
339           }
340           std::cout << "\n";
341
342           std::cout << "proxies:␣";
343           for (size_type i = 0; i < word_size; ++i) {
344       if (low.get(i)) {
345         std::cout << i << ":␣";
346         if (high.get(i)) {
347           std::cout << (*proxies[i].second.root()).element() << "␣";
348         }
349         std::cout << (*proxies[i].first.root()).element();
350         std::cout << "\n";
351       }
352           }
353
354           while (low_bits.size() != 0) {
355       size_type h = low_bits.least_significant_one();
356       if (high_bits.get(h)) {
357         high_bits.unset(h);
358         std::cout << "---␣" << h << "\n";
359         E const* r = proxies[h].second.root();
360         (*r).show_tree();
361       }
362       low_bits.unset(h);
363       std::cout << "---␣" << h << "\n";
364       E const* r = proxies[h].first.root();
365       (*r).show_tree();
366           }
367           std::cout.flush();
368         }
369
370         template <typename M>
371         bool is_valid(M const& mark_store) {
372           if (low.size() == 0) {
373       return true;
374           }
375           bool valid = true;
376           cphstl::bit_store<word_type> high_bits = high;
377           cphstl::bit_store<word_type> low_bits = low;
378           while (low_bits.size() != 0) {
379       size_type h = low_bits.least_significant_one();
380       if (high_bits.get(h)) {
381         high_bits.unset(h);
382         E const* r = proxies[h].second.root();
383         if (! (*r).is_valid(comparator, mark_store)) {
384           std::cerr << "heap-structure␣error:␣" << (*r).element() << "\n";
385           valid = false;
```

```
      }
    }
    low_bits.unset(h);
    E const* r = proxies[h].first.root();
    if (! (*r).is_valid(comparator, mark_store)) {
      std::cerr << "heap-structure error: " << (*r).element() << "\n";
      valid = false;
    }
        }
        if (! valid) {
    return false;
        }
        high_bits = high;
        low_bits = low;
        while (low_bits.size() != 0) {
    size_type h = low_bits.least_significant_one();
    if (high_bits.get(h)) {
      high_bits.unset(h);
      E const* r = proxies[h].second.root();
      if ((*r).height() != h) {
        std::cerr << "height violation: " << (*r).height() << "\n";
        return false;
      }
    }
    low_bits.unset(h);
    E const* r = proxies[h].first.root();
    if ((*r).height() != h) {
      std::cerr << "height violation: " << (*r).height() << "\n";
      return false;
    }
        }
        high_bits = high;
        low_bits = low;
        while (low_bits.size() != 0) {
    size_type h = low_bits.least_significant_one();
    if (high_bits.get(h)) {
      high_bits.unset(h);
      E const* r = proxies[h].second.root();
      if (r == 0 || (*r).owner() != &proxies[h].second) {
        std::cerr << "bit-store violation (high): " << h << "\n";
        return false;
      }
    }
    low_bits.unset(h);
    E const* r = proxies[h].first.root();
    if (r == 0 || (*r).owner() != &proxies[h].first) {
      std::cerr << "bit-store violation (low): " << h << "\n";
      return false;
    }
        }
        return true;
      }

#endif

  protected:

    void insert(E* p, size_type h) {
      if (low.get(h)) {
    high.set(h);
    proxies[h].second.root() = p;
    (*p).owner() = &proxies[h].second;
        }
        else {
    low.set(h);
    proxies[h].first.root() = p;
    (*p).owner() = &proxies[h].first;
        }
      }

      template <typename M>
      void fix(M& mark_store) {
        if (high.size() == 0) {
    return;
```

```
460         }
461         size_type h = high.least_significant_one();
462         E* q = proxies[h].first.root();
463         E* r = proxies[h].second.root();
464         E* s = (*q).join(r, comparator, mark_store);
465         low.unset(h);
466         high.unset(h);
467         insert(s, h + 1);
468       }
469     };
470   }
471
472   #if defined(UNITTEST_PROXY_ARRAY_HEAP_STORE)
473
474   #include "blank-mark-store.h++"
475   #include <cstddef> // std::size_t
476   #include <functional> // std::less
477   #include <iostream>
478   #include <memory>
479   #include <numeric> // std::accumulate
480   #include "weak-heap-node.h++"
481
482   template <typename E, typename H, typename M>
483   void push(E* p, H& heap_store, M& mark_store) {
484     heap_store.inject(p, 0, mark_store);
485   }
486
487   template <typename E, typename C, typename H, typename M>
488   E* extract(C const& comparator, H& heap_store, M& mark_store) {
489     typedef std::size_t size_type;
490     std::pair<E*, size_type> pair = heap_store.eject();
491     E* r = pair.first;
492     size_type h = pair.second;
493     while ((*r).right() != 0) {
494       E* q = (*r).split(comparator);
495       h = h - 1;
496       heap_store.inject(q, h, mark_store);
497     }
498     return r;
499   }
500
501   template <typename T>
502   void test_proxy_array_heap_store() {
503     typedef std::less<T> C;
504     typedef std::allocator<T> A;
505     typedef cphstl::weak_heap_node<T, A> E;
506     typedef cphstl::proxy_array_heap_store<C, A, E> H;
507     typedef cphstl::blank_mark_store<C, A, E> M;
508
509     A allocator;
510     C less;
511     H heap_store(less, allocator);
512     M mark_store(less, allocator);
513     E m(T(4), allocator);
514     E n(T(2), allocator);
515     E o(T(3), allocator);
516     E* p = new E(T(1), allocator);
517
518     std::cout << "push\n";
519     push(&m, heap_store, mark_store);
520     assert(heap_store.is_valid(mark_store));
521
522     std::cout << "push\n";
523     push(&n, heap_store, mark_store);
524     assert(heap_store.is_valid(mark_store));
525
526     std::cout << "push\n";
527     push(&o, heap_store, mark_store);
528     assert(heap_store.is_valid(mark_store));
529
530     std::cout << "push\n";
531     push(p, heap_store, mark_store);
532     assert(heap_store.is_valid(mark_store));
533
```

```
534     std::cout << "extract\n";
535     E* r = extract<E, C, H>(less, heap_store, mark_store);
536     assert(heap_store.is_valid(mark_store));
537
538     std::cout << "push\n";
539     push(r, heap_store, mark_store);
540     assert(heap_store.is_valid(mark_store));
541   }
542
543   int main(int, char**) {
544     test_proxy_array_heap_store<int>();
545     return 0;
546   }
547
548   #endif
549   #endif
```

G.14 proxy-list-heap-store.h++

```
1   /*
2     A stack-like heap store described in the paper by Elmasry et al.
3     [2005]. However, the implementation is simplified using an idea
4     described in the paper by Brodal [1995].
5
6     Author: Asger Bruun, Jyrki Katajainen Â© 2009
7   */
8
9   #ifndef _CPHSTL_DOUBLE_STACK_HEAP_STORE_
10  #define _CPHSTL_DOUBLE_STACK_HEAP_STORE_
11
12  #include <algorithm> // std::swap
13  #include "assert.h++"
14  #include <cmath> // defines ilogb
15  #include <cstddef> // std::size_t
16  #include <iostream>
17  #include <list>
18  #include "heap-proxy.h++"
19  #include <utility> // std::pair
20  #include "weak-heap-node.h++"
21
22  extern int ilogb(double) throw();
23
24  namespace cphstl {
25
26    template<
27      typename C,
28      typename A,
29      typename E,
30      typename P = heap_proxy<E>
31    >
32    class proxy_list_heap_store {
33    public:
34
35      typedef C comparator_type;
36      typedef typename A::template rebind<P>::other proxy_allocator_type;
37      typedef E encapsulator_type;
38      typedef P heap_proxy_type;
39      typedef std::size_t size_type;
40
41    protected:
42
43      comparator_type comparator;
44      proxy_allocator_type proxy_allocator;
45      P* head;
46      P* first_pair;
47      size_type number_of_nodes;
48
49    public:
50
51      proxy_list_heap_store(C const& c = C(), A const& a = A())
52        : comparator(c), proxy_allocator(a), head(0), first_pair(0),
53    number_of_nodes(0) {
54      }
```

```
55
56      ~proxy_list_heap_store() {
57        // Precondition: The heap store must be otherwise empty.
58      }
59
60      size_type size() const {
61        return number_of_nodes;
62      }
63
64      size_type footprint(size_type n) const {
65        return (ilogb(n) + 1) * sizeof(P) + sizeof(proxy_list_heap_store);
66      }
67
68      P* begin() const {
69        return head;
70      }
71
72      P* next(P* current) const {
73        return (*current).successor();
74      }
75
76      E* find_top() const {
77        P* p = begin();
78        if (p == 0) {
79    return 0;
80        }
81        E* top = (*p).root();
82        for (; p != 0; p = (*p).successor()) {
83    E* q = (*p).root();
84    if (comparator((*top).element(), (*q).element())) {
85      top = q;
86    }
87        }
88        return top;
89      }
90
91      template <typename M>
92      void inject(E* p, size_type h, M& mark_store) {
93        assert((*p).is_root());
94        number_of_nodes += 1 << h;
95        if (first_pair != 0) {
96    P* mate = (*first_pair).successor();
97    E* o = (*first_pair).root();
98    E* q = (*o).join((*mate).root(), comparator, mark_store);
99    P* jump = (*first_pair).successor_pair();
100   (*first_pair).update(q, (*first_pair).height() + 1, (*mate).successor(),
101           0);
102   (*mate).update(p, h, head, 0);
103
104   P* s = (*first_pair).successor();
105   if (s != 0 && (*first_pair).height() == (*s).height()) {
106     (*first_pair).successor_pair() = jump;
107   }
108   else {
109     first_pair = jump;
110   }
111
112   if ((*mate).height() == (*head).height()) {
113     (*mate).successor_pair() = first_pair;
114     first_pair = mate;
115   }
116   head = mate;
117   return;
118       }
119       if (head != 0 && (*head).height() == h) {
120   E* q = (*p).join((*head).root(), comparator, mark_store);
121   (*head).update(q, (*head).height() + 1, (*head).successor(), 0);
122   P* s = (*head).successor();
123   if ((s != 0) && ((*head).height() == (*s).height())) {
124     first_pair = head;
125   }
126   return;
127       }
128       P* u = create(p, h, 0, 0);
```

```
129         (*u).successor() = head;
130         head = u;
131       }
132
133       void inject_without_join(E* p, size_type h) {
134         assert((*p).is_root());
135         number_of_nodes += 1 << h;
136         P* u = create(p, h, 0, 0);
137         (*u).successor() = head;
138         head = u;
139       }
140
141       std::pair<E*, size_type> eject() {
142         // Warning: top_ can point to the nodes ejected
143         if (head == 0) {
144     return std::make_pair((E*) 0, (size_type) 0);
145         }
146         P* f = head;
147         E* r = (*f).root();
148         head = (*head).successor();
149         if (first_pair == f) {
150     first_pair = (*f).successor_pair();
151     (*f).successor_pair() = 0;
152         }
153         number_of_nodes -= 1 << (*f).height();
154         std::pair<E*, size_type> answer = std::make_pair(r, (*f).height());
155         destroy(f);
156         (*r).owner() = (P*) 0;
157         return answer;
158       }
159
160       template <typename M>
161       void concatenate(E* q, size_type h, M& mark_store) {
162         while (q != 0) {
163     mark_store.unmark(q);
164     E* s = (*q).release_subheap();
165     inject_without_join(q, h);
166     h = h - 1;
167     q = s;
168         }
169       }
170
171       void replace(E* new_root) {
172         heap_proxy_type* proxy = (heap_proxy_type*) (*new_root).owner();
173         (*proxy).root() = new_root;
174         (*new_root).owner() = (void *) proxy;
175       }
176
177       template <typename M>
178       void meld(proxy_list_heap_store& other, M& mark_store) {
179         // Precondition: The comparators and allocators must be compatible.
180         std::list<P*, A> tmp;
181         size_type n = number_of_nodes;
182         size_type m  = other.number_of_nodes;
183         P* s = begin();
184         P* t = other.begin();
185         while (s != 0 && t != 0) {
186     if ((*s).height() < (*t).height()) {
187       tmp.push_front(s);
188       s = (*s).successor();
189       pop();
190     }
191     else {
192       tmp.push_front(t);
193       t = (*t).successor();
194       other.pop();
195     }
196         }
197         if (begin() == 0) {
198     swap(other);
199         }
200         while (! tmp.empty()) {
201     P* t = tmp.front();
202     E* r = (*t).root();
```

```
203     inject(r, (*t).height(), mark_store);
204     destroy(t);
205     tmp.pop_front();
206         }
207         number_of_nodes = n + m;
208         other.number_of_nodes = 0;
209       }
210
211       void swap(proxy_list_heap_store& other) {
212         // Precondition: The comparators and allocators must be compatible.
213         std::swap(head, other.head);
214         std::swap(first_pair, other.first_pair);
215         std::swap(number_of_nodes, other.number_of_nodes);
216       }
217
218   #ifdef DEBUG
219
220       void show() {
221         std::cout << "==========\n";
222         for (P* p = begin(); p != 0; p = (*p).successor()) {
223     std::cout << "---" << (*p).height() << "\n";
224     E* r = (*p).root();
225     (*r).show_tree();
226         }
227         std::cout << "==========\n";
228         std::cout.flush();
229       }
230
231       template <typename M>
232       bool is_valid(M const& mark_store) {
233         P const* p = begin();
234         if (p == 0) {
235     return true;
236         }
237         for (; p != 0; p = (*p).successor()) {
238     bool valid = true;
239     E const* r = (*p).root();
240     for (; r != 0; r = (*r).successor()) {
241       if (! (*r).is_valid(comparator, mark_store)) {
242         std::cerr << "heap-structure error: " << (*r).element() << "\n";
243         valid = false;
244       }
245     }
246     if (! valid) {
247       return false;
248     }
249         }
250         p = begin();
251         P const* q = (*p).successor();
252         for (; q != 0; p = q, q = (*p).successor()) {
253     if ((*p).height() > (*q).height()) {
254       std::cerr << "height violation: " << (*p).height() << "\n";
255       return false;
256     }
257         }
258         p = begin();
259         q = (*p).successor();
260         for (; q != 0; p = q, q = (*p).successor()) {
261     if ((*p).successor_pair() != 0 && (*p).height() != (*q).height()) {
262       std::cerr << "pair violation: " << (*p).height() << "\n";
263       return false;
264     }
265         }
266         p = begin();
267         q = (*p).successor();
268         for (; q != 0; p = q, q = (*p).successor()) {
269     E const* r = (*p).root();
270     if (! (*r).is_root()) {
271       std::cerr << "root violation: " << (*p).height() << "\n";
272       return false;
273     }
274     if ((*r).owner() != p) {
275       std::cerr << "owner violation: " << (*p).height() << "\n";
276       return false;
```

```
277     }
278         }
279         return true;
280       }
281
282   #endif
283
284     protected:
285
286       P* create(E* r, size_type height, P* next, P* next_pair) {
287         assert((*r).is_root());
288         P* u = proxy_allocator.allocate(1);
289         new (u) P(r, height, next, next_pair);
290         (*r).owner() = u;
291         return u;
292       }
293
294       void destroy(P* u) {
295         assert(u != 0);
296         (*u).~P();
297         proxy_allocator.deallocate(u, 1);
298       }
299
300       void pop() {
301         P* f = head;
302         head = (*head).successor();
303         if (first_pair == f) {
304     first_pair = (*f).successor_pair();
305     (*f).successor_pair() = 0;
306         }
307       }
308     };
309   }
310
311   #if defined(UNITTEST_PROXY_LIST_HEAP_STORE)
312
313   #include "blank-mark-store.h++"
314   #include <cstddef> // std::size_t
315   #include <functional>
316   #include <iostream>
317   #include <memory>
318   #include <numeric> // std::accumulate
319   #include "weak-heap-node.h++"
320
321   template <typename E, typename H, typename M>
322   void push(E* p, H& heap_store, M& mark_store) {
323     heap_store.inject(p, 0, mark_store);
324   }
325
326   template <typename E, typename C, typename H, typename M>
327   E* extract(C const& comparator, H& heap_store, M& mark_store) {
328     typedef std::size_t size_type;
329     std::pair<E*, size_type> pair = heap_store.eject();
330     E* r = pair.first;
331     size_type h = pair.second;
332     while ((*r).right() != 0) {
333       E* q = (*r).split(comparator);
334       h = h - 1;
335       heap_store.inject(q, h, mark_store);
336     }
337     return r;
338   }
339
340   template <typename T>
341   void proxy_list_heap_store_test() {
342     typedef std::allocator<T> A;
343     typedef std::less<T> C;
344     typedef cphstl::weak_heap_node<T, A> E;
345     typedef cphstl::heap_proxy<E> P;
346     typedef cphstl::proxy_list_heap_store<C, A, E, P> H;
347     typedef cphstl::blank_mark_store<C, A, E> M;
348
349     A allocator;
350     C less;
```

```
351   H heap_store(less, allocator);
352   M mark_store(less, allocator);
353   E m(T(4), allocator);
354   E n(T(2), allocator);
355   E o(T(3), allocator);
356   E* p = new E(T(1), allocator);
357
358   std::cout << "push ";
359   push(&m, heap_store, mark_store);
360   heap_store.show();
361   std::cout << "push ";
362   push(&n, heap_store, mark_store);
363   heap_store.show();
364   std::cout << "push ";
365   push(&o, heap_store, mark_store);
366   heap_store.show();
367   std::cout << "push ";
368   push(p, heap_store, mark_store);
369   heap_store.show();
370   assert(heap_store.is_valid(mark_store));
371   std::cout << "validated\n";
372
373   std::cout << "extract ";
374   E* r = extract<E, C, H>(less, heap_store, mark_store);
375   heap_store.show();
376
377   std::cout << "push ";
378   push(r, heap_store, mark_store);
379   assert(heap_store.is_valid(mark_store));
380   std::cout << "validated\n";
381 }
382
383 template <typename T>
384 void iteration_test() {
385   typedef std::allocator<T> A;
386   typedef std::less<T> C;
387   typedef cphstl::weak_heap_node<T, A> E;
388   typedef cphstl::heap_proxy<E> P;
389   typedef cphstl::proxy_list_heap_store<C, A, E, P> H;
390   typedef cphstl::blank_mark_store<C, A, E> M;
391
392   A allocator;
393   C less;
394   T a[] = {8, 10, 12, 1, 6, 5, 3, 7};
395   unsigned int k = sizeof(a) / sizeof(T);
396   H heap_store(less, allocator);
397   M mark_store(less, allocator);
398   for (unsigned int j = 0; j < k; ++j) {
399     E* r = new E(a[j], allocator);
400     push(r, heap_store, mark_store);
401   }
402
403   T sum = T(0);
404   for (P const* h = heap_store.begin(); h != 0; h = (*h).successor()) {
405     for (E const* p = (*h).root(); p != 0; p = (*p).successor()) {
406       sum += (*p).element();
407     }
408   }
409   assert(sum == std::accumulate(&a[0], &a[k], T(0)));
410 }
411
412 int main(int, char**) {
413   proxy_list_heap_store_test<int>();
414   iteration_test<int>();
415   return 0;
416 }
417
418 #endif
419 #endif
```

G.15 proxy-list-node.h++

```
1 /*
```

```
2     A heap proxy maintains the height and a pointer to the root
3     of each heap.
4
5     Author: Jyrki Katajainen Â© 2009
6   */
7
8   #ifndef _CPHSTL_HEAP_PROXY_
9   #define _CPHSTL_HEAP_PROXY_
10
11  #include "assert.h++"
12  #include <cstddef> // std::size_t
13
14  namespace cphstl {
15
16    template <typename E>
17    class heap_proxy {
18    public:
19
20      typedef E encapsulator_type;
21      typedef std::size_t size_type;
22
23      heap_proxy(E* root, size_type height = 0,
24            heap_proxy* next = 0,
25            heap_proxy* next_pair = 0)
26        : root_(root), height_(height), next_(next), next_pair_(next_pair) {
27      }
28
29      ~heap_proxy() {
30      }
31
32      void update(E* root, size_type height, heap_proxy* next,
33      heap_proxy* next_pair) {
34        assert(this != 0);
35        assert((*root).is_root());
36        root_ = root;
37        height_ = height;
38        next_ = next;
39        next_pair_ = next_pair;
40        (*root).owner() = this;
41      }
42
43      E* root() const {
44        return root_;
45      }
46
47      E*& root() {
48        return root_;
49      }
50
51      size_type height() const {
52        return height_;
53      }
54
55      size_type& height() {
56        return height_;
57      }
58
59      heap_proxy* successor() const {
60        return next_;
61      }
62
63      heap_proxy*& successor() {
64        return next_;
65      }
66
67      heap_proxy* successor_pair() const {
68        return next_pair_;
69      }
70
71      heap_proxy*& successor_pair() {
72        return next_pair_;
73      }
74
75    protected:
```

```
76
77      E* root_;
78      size_type height_;
79      heap_proxy* next_;
80      heap_proxy* next_pair_;
81
82    private:
83
84      heap_proxy();
85      heap_proxy(heap_proxy const&);
86      heap_proxy& operator = (heap_proxy const&);
87
88    };
89  }
90
91  #if defined(UNITTEST_HEAP_PROXY)
92
93  #include <memory> // std::allocator
94  #include "weak-heap-node.h++"
95
96  template <typename T>
97  void test_heap_proxy() {
98    typedef std::allocator<T> A;
99    typedef cphstl::weak_heap_node<T, A> N;
100   typedef cphstl::heap_proxy<N> P;
101   N* dummy = new N(T(0), A());
102   P* p = new P(dummy);
103   P* q = new P(dummy);
104   P* r = new P(dummy);
105
106   (*p).successor() = q;
107   (*q).successor() = r;
108   (*r).successor() = 0;
109
110   assert((*p).height() == 0);
111   assert((*q).root() == dummy);
112   assert((*r).successor_pair() == 0);
113   assert((*p).successor() == q);
114   assert((*q).successor() == r);
115   assert((*r).successor() == 0);
116
117   (*r).height() = 4;
118   assert((*r).height() == 4);
119 }
120
121 int main(int, char**) {
122   test_heap_proxy<int>();
123   test_heap_proxy<char>();
124   return 0;
125 }
126
127 #endif
128 #endif
```

G.16 root-list-heap-store.h++

```
1   /*
2     A root-list heap store; the redundant binary system is used for
3     keeping track of the roots.
4
5     Authors: Asger Bruun, Stefan Edelkamp, Jyrki Katajainen Â© 2010
6   */
7
8   #ifndef _CPHSTL_ROOT_LIST_HEAP_STORE_
9   #define _CPHSTL_ROOT_LIST_HEAP_STORE_
10
11  #include <algorithm> // std::swap
12  #include "assert.h++"
13  #include "bit-store.h++"
14  #include <cmath> // ilogb
15  #include <cstddef> // std::size_t
16  #include <iostream>
17  #include <list>
```

```
18 #include <utility> // std::pair
19
20 extern int ilogb(double) throw();
21
22 namespace cphstl {
23
24   template<
25     typename C,
26     typename A,
27     typename E,
28     typename W = unsigned long
29   >
30   class root_list_heap_store {
31   public:
32
33     typedef C comparator_type;
34     typedef A allocator_type;
35     typedef E encapsulator_type;
36     typedef E component_type;
37     typedef W word_type;
38     typedef std::size_t size_type;
39
40   protected:
41
42     comparator_type comparator;
43     std::list<E*, A> two_stack;
44     E* head;
45     cphstl::bit_store<word_type> high;
46     cphstl::bit_store<word_type> low;
47
48   public:
49
50     root_list_heap_store(C const& c = C(), A const& a = A())
51       : comparator(c), two_stack(a), head(0), high(), low() {
52     }
53
54     ~root_list_heap_store() {
55       // Precondition: The heap store must be otherwise empty.
56     }
57
58     size_type footprint(size_type n) const {
59       return (ilogb(n) + 1) * 3 + sizeof(root_list_heap_store);
60     }
61
62     E* find_top() const {
63       E* p = head;
64       if (p == 0) {
65   return 0;
66       }
67       E* top = p;
68       for (p = (*p).parent(); p != 0; p = (*p).parent()) {
69   if (comparator((*top).element(), (*p).element())) {
70     top = p;
71   }
72       }
73       return top;
74     }
75
76     void inject(E* p, size_type h) {
77       assert(p != 0);
78       (*p).parent() = head;
79       if (head != 0) {
80   (*head).left() = p;
81       }
82       (*p).left() = 0;
83       head = p;
84       if (low.get(h)) {
85   high.set(h);
86   two_stack.push_front(p);
87       }
88       else {
89   low.set(h);
90       }
91       if (two_stack.empty()) {
```

```
92      return;
93          }
94          h = high.least_significant_one();
95          E* q = two_stack.front();
96          two_stack.pop_front();
97          p = (*q).left();
98          (*q).left() = 0;
99          E* r = (*q).parent();
100         (*r).left() = 0;
101         E* t = (*r).parent();
102         E* s = (*q).join(r, comparator);
103         (*s).left() = p;
104         if (p != 0) {
105     (*p).parent() = s;
106         }
107         else {
108     head = s;
109         }
110         (*s).parent() = t;
111         if (t != 0) {
112     (*t).left() = s;
113         }
114         low.unset(h);
115         high.unset(h);
116         if (low.get(h + 1)) {
117     high.set(h + 1);
118     two_stack.push_front(s);
119         }
120         else {
121     low.set(h + 1);
122         }
123       }
124
125       std::pair<E*, size_type> eject() {
126         // Warning: top_ can point to the nodes ejected
127         if (head == 0) {
128     return std::make_pair((E*) 0, (size_type) 0);
129         }
130         size_type h = low.least_significant_one();
131         if (high.get(h)) {
132     high.unset(h);
133     two_stack.pop_front();
134         }
135         E* r = head;
136         head = (*head).parent();
137         (*r).parent() = 0;
138         (*head).left() = 0;
139         return std::make_pair(r, h);
140       }
141
142       template <typename M>
143       void concatenate(E* q, word_type h, M& mark_store) {
144         while (q != 0) {
145     mark_store.unmark(q);
146     E* s = (*q).release_subheap();
147     inject(q, h);
148     h = h - 1;
149     q = s;
150         }
151       }
152
153       void meld(root_list_heap_store& other) {
154         // Precondition: The comparators and allocators must be compatible.
155         std::list<std::pair<E*, size_type>, A> tmp;
156         std::pair<E*, size_type> s = eject();
157         std::pair<E*, size_type> t = other.eject();
158         while (s.first != 0 && t.first != 0) {
159     if (s.second < t.second) {
160       tmp.push_front(s);
161       s = eject();
162     }
163     else {
164       tmp.push_front(t);
165       t = other.eject();
```

```
166     }
167         }
168         if (head == 0) {
169     swap(other);
170         }
171         while (! tmp.empty()) {
172     t = tmp.front();
173     inject(t.first, t.second);
174     tmp.pop_front();
175         }
176       }
177 
178       void swap(root_list_heap_store& other) {
179         // Precondition: The comparators and allocators must be compatible.
180         std::swap(two_stack, other.two_stack);
181         std::swap(head, other.head);
182         std::swap(high, other.high);
183         std::swap(low, other.low);
184       }
185 
186   #ifdef DEBUG
187 
188       void show() {
189         std::cout << "␣==␣==␣==␣␣=␣\n";
190         cphstl::bit_store<word_type> high_bits = high;
191         cphstl::bit_store<word_type> low_bits = low;
192         for (E* p = head; p != 0; p = (*p).parent()) {
193     size_type h = low_bits.least_significant_one();
194     if (high_bits.get(h)) {
195       high_bits.unset(h);
196     }
197     else {
198       low_bits.unset(h);
199     }
200     std::cout << "---␣" << h << "\n";
201     (*p).show_tree();
202         }
203         std::cout << "␣==␣==␣==␣␣=␣\n";
204         std::cout.flush();
205       }
206 
207       template <typename M>
208       bool is_valid(M const& mark_store) {
209         E const* p = head;
210         if (p == 0) {
211     return true;
212         }
213         for (; p != 0; p = (*p).parent()) {
214     bool valid = true;
215     E const* r = (*p).parent();
216     for (; p != r; p = (*p).successor()) {
217       if (! (*p).is_valid(comparator, mark_store)) {
218         std::cerr << "heap-structure␣error:␣" << (*p).element() << "\n";
219         valid = false;
220       }
221     }
222     if (! valid) {
223       return false;
224     }
225         }
226         cphstl::bit_store<word_type> high_bits = high;
227         cphstl::bit_store<word_type> low_bits = low;
228         p = head;
229         E const* q = (*p).parent();
230         for (; q != 0; p = q, q = (*p).parent()) {
231     if ((*p).height() > (*q).height()) {
232       std::cerr << "height␣violation:␣" << (*p).height() << "\n";
233       return false;
234     }
235     word_type h = low_bits.least_significant_one();
236     if (high_bits.get(h)) {
237       high_bits.unset(h);
238     }
239     else {
```

```
240       low_bits.unset(h);
241     }
242     if ((*p).height() != h) {
243       std::cerr << "bit-store violation: " << h << "\n";
244       return false;
245     }
246         }
247         p = head;
248         typename std::list<E*, A>::const_iterator i = two_stack.begin();
249         q = (*p).parent();
250         E* r = *i;
251         for (; q != 0; p = q, q = (*p).parent()) {
252     word_type h = (*p).height();
253     if (! low.get(h)) {
254       std::cerr << "bit-store violation (low): " << h << "\n";
255       return false;
256     }
257     if (p == r) {
258       if (h != (*q).height()) {
259         std::cerr << "pair violation: " << h << "\n";
260         return false;
261       }
262       if (! high.get(h)) {
263         std::cerr << "bit-store violation (high): " << h << "\n";
264         return false;
265       }
266       ++i;
267       r = *i;
268     }
269         }
270         return true;
271       }
272
273   #endif
274
275     };
276   }
277
278   #if defined(UNITTEST_ROOT_LIST_HEAP_STORE)
279
280   #include "blank-mark-store.h++"
281   #include <cstddef> // std::size_t
282   #include <functional>
283   #include <iostream>
284   #include <memory>
285   #include <numeric> // std::accumulate
286   #include "weak-heap-node.h++"
287
288   template <typename E, typename H>
289   void push(E* p, H& heap_store) {
290     heap_store.inject(p, 0);
291   }
292
293   template <typename E, typename C, typename H>
294   E* extract(C const& comparator, H& heap_store) {
295     typedef std::size_t size_type;
296     std::pair<E*, size_type> pair = heap_store.eject();
297     E* r = pair.first;
298     size_type h = pair.second;
299     while ((*r).right() != 0) {
300       E* q = (*r).split(comparator);
301       h = h - 1;
302       heap_store.inject(q, h);
303     }
304     return r;
305   }
306
307   template <typename T>
308   void test_root_list_heap_store() {
309     typedef std::allocator<T> A;
310     typedef std::less<T> C;
311     typedef cphstl::weak_heap_node<T, A> E;
312     typedef cphstl::root_list_heap_store<C, A, E> H;
313     typedef cphstl::blank_mark_store<C, A, E> M;
```

```
314
315     A allocator;
316     C less;
317     H heap_store(less, allocator);
318     M mark_store(less, allocator);
319     E m(T(4), allocator);
320     E n(T(2), allocator);
321     E o(T(3), allocator);
322     E* p = new E(T(1), allocator);
323
324     std::cout << "push\n";
325     push(&m, heap_store);
326     heap_store.show();
327
328     std::cout << "push\n";
329     push(&n, heap_store);
330     heap_store.show();
331
332     std::cout << "push\n";
333     push(&o, heap_store);
334     heap_store.show();
335
336     std::cout << "push␣";
337     push(p, heap_store);
338     heap_store.show();
339     assert(heap_store.is_valid(mark_store));
340     std::cout << "validated\n";
341
342     std::cout << "extract\n";
343     E* r = extract<E, C, H>(less, heap_store);
344     heap_store.show();
345
346     std::cout << "push␣";
347     push(r, heap_store);
348     assert(heap_store.is_valid(mark_store));
349     std::cout << "validated\n";
350   }
351
352   int main(int, char**) {
353     test_root_list_heap_store<int>();
354     return 0;
355   }
356
357   #endif
358   #endif
```

G.17 simple-mark-store.h++

```
1   /*
2     A simplified eager mark store. The invariants maintained:
3
4     1) A marked node does not have any neighbouring marked node
5     (parent, child, or sibling).
6
7     2) A marked node is always a right child of its parent.
8
9     Authors: Stefan Edelkamp, Jyrki Katajainen Â© 2009, 2010
10  */
11
12  #ifndef _CPHSTL_SIMPLE_MARK_STORE_
13  #define _CPHSTL_SIMPLE_MARK_STORE_
14
15  #include <algorithm> // std::max, std::swap
16  #include "assert.h++"
17  #include "bit-store.h++"
18  #include "comparator-proxy.h++"
19  #include <cstddef> // std::size_t
20  #include <cmath> // ilogb
21  #include <iostream>
22  #include <vector>
23
24  extern int ilogb(double) throw();
25
```

```
26  namespace cphstl {
27
28    template <typename C, typename A, typename E>
29    class simple_mark_store {
30    public:
31
32      typedef C comparator_type;
33      typedef A allocator_type;
34      typedef E encapsulator_type;
35      typedef typename E::mark_type mark_type;
36      typedef typename E::height_type height_type;
37      typedef typename E::index_type index_type;
38      typedef unsigned long word_type;
39      typedef std::size_t size_type;
40
41    protected:
42
43      comparator_proxy<C> comparator;
44      std::vector<E*, A> nodes;
45      std::vector<bit_store<word_type>, A> marks;
46      bit_store<word_type> teams;
47      E* buffered_mark;
48
49    private:
50
51      simple_mark_store(simple_mark_store const&);
52      simple_mark_store& operator = (simple_mark_store const&);
53
54    public:
55
56      simple_mark_store(C const& c = C(), A const& a = A())
57        : comparator(c), nodes(a), marks(a), teams(), buffered_mark(0) {
58        marks.resize(bit_store<word_type>::word_size, bit_store<word_type>());
59      }
60
61      ~simple_mark_store() {
62      }
63
64      size_type footprint(size_type n) const {
65        return (ilogb(n) + 1) * 6 * (sizeof(E*) + sizeof(word_type)) +
66    2 * sizeof(word_type);
67      }
68
69      E* find_top() const {
70        if (nodes.size() == 0) {
71    return 0;
72        }
73        E* top = nodes[0];
74        for (index_type i = 1; i < index_type(nodes.size()); ++i) {
75    if (comparator((*top).element(), (*nodes[i]).element())) {
76      top = nodes[i];
77    }
78        }
79        return top;
80      }
81
82      bool is_marked(E const* p) const {
83        return (*p).type() != E::unmarked;
84      }
85
86      void mark(E* q) {
87        //      std::cout << "mark: " << (*q).element() << "\n";
88        assert(q != 0);
89        if (is_marked(q) || (*q).is_root()) {
90    return;
91        }
92        assert(buffered_mark == 0);
93        insert_node(q);
94        insert_mark(q);
95        buffered_mark = q;
96      }
97
98      void unmark(E* q) {
99        if (! is_marked(q)) {
```

```
100     return;
101         }
102         if (buffered_mark == q) {
103     buffered_mark = 0;
104         }
105         remove_mark(q);
106         remove_node(q);
107       }
108
109       template <typename H>
110       void reduce(H& heap_store) {
111         //       std::cout << "reduce:\n";
112         size_type n = heap_store.size();
113         if (n == 0) {
114     return;
115         }
116         if (buffered_mark != 0) {
117     E* r = (*buffered_mark).right();
118     if (r != 0 && is_marked(r)) {
119       buffered_mark = parent_transformation(r);
120     }
121     propagate_up(buffered_mark, heap_store);
122     assert(is_valid());
123         }
124         if (nodes.size() > size_type(ilogb(n) + 1)) {
125     buffered_mark = team_transformation();
126     if ((*buffered_mark).is_root()) {
127       heap_store.replace(buffered_mark);
128     }
129     else {
130       propagate_up(buffered_mark, heap_store);
131     }
132     assert(is_valid());
133         }
134         buffered_mark = 0;
135         //       heap_store.show();
136         //       show();
137       }
138
139       template <typename H>
140       void meld(simple_mark_store& other, H& heap_store) {
141         index_type k = other.nodes.size();
142         for (index_type i = 0; i != k; ++i) {
143     E* p = other.nodes[i];
144     other.remove_mark(p);
145     other.remove_node(p);
146     insert_node(p);
147     insert_mark(p);
148     reduce(heap_store);
149         }
150         other.nodes.resize(0);
151         other.marks.resize(0);
152       }
153
154       void swap(simple_mark_store& other) {
155         // Precondition: The comparators are compatible.
156         std::swap(nodes, other.nodes);
157         std::swap(marks, other.marks);
158         std::swap(teams, other.teams);
159       }
160
161   #ifdef DEBUG
162
163       bool is_valid() const {
164         bool valid_1 = true;
165         for (index_type i = 0; i < index_type(nodes.size()); ++i) {
166     E const* p = nodes[i];
167     bool okei = (*p).index() == i && is_marked(p);
168     if (! okei) {
169       std::cout << "error: incorrect handle (" << (*p).element() << ")\n";
170     }
171     okei = ! (*p).is_root();
172     if (! okei) {
173       std::cout << "error: a marked root (" << (*p).element() << ")\n";
```

```
174     }
175     valid_1 &= okei;
176         }
177         bool valid_2 = true;
178         bit_store<word_type> temp = teams;
179         word_type one_bits = temp.size();
180         for (word_type j = 0; j < one_bits; ++j) {
181     height_type h = temp.choose();
182     temp.unset(h);
183     bool okei = marks[h].size() > 1;
184     valid_2 &= okei;
185     if (! okei) {
186       std::cout << "error: not a team " << int(h) << "\n";
187     }
188         }
189         bool valid_3 = true;
190         for (height_type h = 0; h < height_type(marks.size()); ++h) {
191     temp = marks[h];
192     word_type one_bits = temp.size();
193     for (word_type i = 0; i < one_bits; ++i) {
194       index_type j = temp.choose();
195       temp.unset(j);
196       E const* p = nodes[j];
197       bool okei = is_marked(p);
198       valid_3 &= okei;
199       if (! okei) {
200         std::cout << "error: extra mark (" << (*p).element() << ")\n";
201       }
202     }
203         }
204         bool valid_4 = true;
205         for (index_type i = 0; i < index_type(nodes.size()); ++i) {
206     E const* p = nodes[i];
207     bool okei = marks[(*p).height()].get((*p).index());
208     valid_4 &= okei;
209     if (! okei) {
210       std::cout << "error: missing mark   (" << (*p).element() << ")\n";
211     }
212         }
213         return valid_1 && valid_2 && valid_3 && valid_4;
214       }
215
216       void show() const {
217         std::cout << "marked elements:\n  ";
218         for (index_type i = 0; i < index_type(nodes.size()); ++i) {
219     std::cout << (*nodes[i]).element() << " ";
220         }
221         std::cout << "\n";
222
223         std::cout << "teams:\n  ";
224         for (height_type j = 0; j < height_type(teams.size()); ++j) {
225     if (teams.get(j)) {
226       std::cout << int(j) << " ";
227     }
228         }
229         std::cout << "\n";
230
231         std::cout << "marked elements:\n";
232         for (height_type h = 0; h < height_type(marks.size()); ++h) {
233     bit_store<word_type> temp = marks[h];
234     index_type one_bits = temp.size();
235     if (one_bits != 0) {
236       std::cout << "  " << int(h) << ": ";
237     }
238     for (index_type i = 0; i < one_bits; ++i) {
239       index_type j = temp.choose();
240       temp.unset(j);
241       E const* p = nodes[j];
242       std::cout << (*p).element() << " ";
243     }
244     if (one_bits != 0) {
245       std::cout << "\n";
246     }
247         }
```

```
248       }
249
250   #endif
251
252     protected:
253
254       void insert_node(E* q) {
255         nodes.push_back(q);
256         (*q).index() = nodes.size() - 1;
257       }
258
259       void insert_mark(E* q) {
260         // Precondition: q must have its new height
261         assert(! is_marked(q));
262         marks[(*q).height()].set((*q).index());
263         if (marks[(*q).height()].size() > 1) {
264     teams.set((*q).height());
265         }
266         (*q).type() = E::singleton;
267       }
268
269       void remove_mark(E* r) {
270         // Precondition: r must have its old height
271         marks[(*r).height()].unset((*r).index());
272         if (marks[(*r).height()].size() < 2) {
273     teams.unset((*r).height());
274         }
275         (*r).type() = E::unmarked;
276       }
277
278       void remove_node(E* x) {
279         // Precondition: Marks of all other nodes must be valid.
280         E* y = nodes.back();
281         remove_mark(y);
282         nodes.pop_back();
283         (*y).index() = -1;
284         if (x != y) {
285     index_type i = (*x).index();
286     nodes[i] = y;
287     (*x).index() = -1;
288     (*y).index() = i;
289     insert_mark(y);
290         }
291       }
292
293       template <typename H>
294       void propagate_up(E* q, H& heap_store) {
295         while (true) {
296     E* p = (*q).parent();
297     if ((*p).right() == q) {
298       if (is_marked(p)) {
299         q = parent_transformation(q);
300         if ((*q).is_root()) {
301           heap_store.replace(q);
302         }
303         return;
304       }
305       return;
306     }
307     assert((*p).left() == q);
308     if (is_marked(p)) {
309       q = left_child_transformation(p);
310       if ((*q).is_root()) {
311         heap_store.replace(q);
312         return;
313       }
314       continue;
315     }
316     if (is_marked((*p).right())) {
317       q = sibling_transformation(q);
318       continue;
319     }
320     cleaning_transformation(q);
321     return;
```

```
322       }
323     }
324
325   public:
326
327     /*
328             p                       p
329           /   \                   /   \
330         [q]    r       ->       r     [q]
331         / \   / \               / \   / \
332        a   b c   d             a   d c   b
333     */
334
335     void cleaning_transformation(E* q) {
336       //       std::cout << "cleaning: " << (*q).element() << "\n";
337       //       show();
338       assert(is_marked(q));
339       E* p = (*q).parent();
340       assert(! is_marked(p));
341       assert((*p).left() == q);
342       E* r = (*p).right();
343       assert(! is_marked(r));
344       (*p).left() = r;
345       (*p).right() = q;
346       E* a = (*q).left();
347       E* c = (*r).left();
348       (*q).left() = c;
349       (*r).left() = a;
350       if (a != 0) {
351   (*a).parent() = r;
352       }
353       if (c != 0) {
354   (*c).parent() = q;
355       }
356       assert(is_valid());
357     }
358
359     /*
360           g                       g                       g
361         \?/                       |                       |
362           p                     [q]                     [r]
363         /   \                   /   \                   /   \
364       [q]   [r]      ->       p       r       or      p       q
365       / \   / \               / \   / \               / \   / \
366      a   b c   d             a   c b   d             a   c d   b
367
368     */
369
370     E* sibling_transformation(E* q) {
371       //       std::cout << "sibling: " << (*q).element() << "\n";
372       //       show();
373       assert(q != 0);
374       E* p = (*q).parent();
375       assert(! is_marked(p));
376       assert((*p).left() == q);
377       assert(is_marked(q));
378       E* r = (*p).right();
379       assert(is_marked(r));
380       E* a = (*q).left();
381       E* b = (*q).right();
382       E* c = (*r).left();
383       E* d = (*r).right();
384       E* g = (*p).parent();
385       remove_mark(q);
386       remove_mark(r);
387       if (comparator((*q).element(), (*r).element())) {
388   (*p).height() -= 1;
389   (*r).height() += 1;
390   (*r).parent() = g;
391   if ((*g).right() == p) {
392     (*g).right() = r;
393   }
394   else {
395     (*g).left() = r;
```

```
396     }
397     (*r).left() = p;
398     (*r).right() = q;
399     (*p).parent() = r;
400     (*q).parent() = r;
401     (*p).left() = a;
402     (*p).right() = c;
403     (*q).left() = b;
404     (*q).right() = d;
405     if (a != 0) {
406       (*a).parent() = p;
407       (*c).parent() = p;
408       (*b).parent() = q;
409       (*d).parent() = q;
410     }
411     insert_mark(r);
412     remove_node(q);
413     return r;
414         }
415         (*p).height() -= 1;
416         (*q).height() += 1;
417         (*q).parent() = g;
418         if ((*g).right() == p) {
419     (*g).right() = q;
420         }
421         else {
422     (*g).left() = q;
423         }
424         (*q).left() = p;
425         (*q).right() = r;
426         (*p).parent() = q;
427         (*r).parent() = q;
428         (*p).left() = a;
429         (*p).right() = c;
430         (*r).left() = b;
431         (*r).right() = d;
432         if (a != 0) {
433     (*a).parent() = p;
434     (*c).parent() = p;
435     (*b).parent() = r;
436     (*d).parent() = r;
437         }
438         insert_mark(q);
439         remove_node(r);
440         return q;
441       }
442
443     protected:
444
445       /*
446            g                    g                    g
447            |                    |                    |
448            p                   [r]                   p
449          /   \                /   \                /   \
450         a    [q]     ->      a    [q]      or     a    [q]     plus parent
451              / \                  / \                  / \     transformation
452            [r]  s               p   s                r   s    at q
453            / \  / \             / \ / \              / \ / \
454           b  c  d  e           c  b d  e            b  c d  c
455       */
456
457       E* left_child_transformation(E* q) {
458         //       std::cout << "left_child: " << (*q).element() << "\n";
459         //       show();
460         assert(is_marked(q));
461         E* p = (*q).parent();
462         assert(! is_marked(p));
463         assert(q == (*p).right());
464         E* r = (*q).left();
465         assert(is_marked(r));
466         E* s = (*q).right();
467         assert(! is_marked(s));
468         assert(is_valid());
469         if (comparator((*p).element(), (*r).element())) {
```

```
    remove_mark(r);
    r = (*r).promote(p);
    E* b = (*p).left();
    E* c = (*p).right();
    (*p).left() = c;
    (*p).right() = b;
    insert_mark(r);
    if ((*r).is_root()) {
      remove_mark(r);
      remove_node(r);
    }
    q = parent_transformation(q);
    return q;
        }
        remove_mark(r);
        remove_node(r);
        q = parent_transformation(q);
        return q;
      }

      /*
           (p)                  [q]                  (p)
          /   \                /   \                /   \
         a     [q]     ->     a     p       or     a     q
              /   \                /   \                /   \
             b     c              c     b              b     c

      */

      E* parent_transformation(E* q) {
        std::cout << "parent: " << (*q).element() << "\n";
        assert(is_marked(q));
        E* p = (*q).parent();
        assert((*p).right() == q);
        assert(is_valid());
        if (is_marked(p)) {
    remove_mark(p);
    remove_mark(q);
    if (comparator((*p).element(), (*q).element())) {
      (*p).swap_neighbours(q);
      insert_mark(q);
      remove_node(p);
      assert(is_valid());
      return q;
    }
    std::cout << "remove: " << (*q).element() << "\n";
    insert_mark(p);
    remove_node(q);
    assert(is_valid());
    return p;
        }
        remove_mark(q);
        if (comparator((*p).element(), (*q).element())) {
    (*p).swap_neighbours(q);
    insert_mark(q);
    if ((*q).is_root()) {
      remove_mark(q);
      remove_node(q);
    }
    assert(is_valid());
    return q;
        }
        remove_node(q);
        assert(is_valid());
        return p;
      }

      /*
        p          r              p          r       [q]        [s]
         \          \              \          \        \          \
          [q]   ,    [s]   ->       r    or    p   ,    s    or    q
          / \        / \            / \        / \      / \        / \
         a   b      c   d          a   c      c   a    b   d      d   b
      */
```

```
544
545       E* pair_action(E* p, E* r, E* q, E* s) {
546         // Precondition: p < r, q < s
547         E* a = (*q).left();
548         E* b = (*q).right();
549         E* c = (*s).left();
550         E* d = (*s).right();
551         E* g = (*p).parent();
552         (*r).right() = p;
553         (*p).parent() = r;
554         (*p).left() = c;
555         (*p).right() = b;
556         if (b != 0) {
557       (*c).parent() = p;
558       (*a).parent() = p;
559         }
560         (*p).height() -= 1;
561         if (g != 0) {
562       if ((*g).right() == p) {
563         (*g).right() = s;
564       }
565       else {
566         (*g).left() = s;
567       }
568         }
569         (*s).parent() = g;
570         (*s).right() = q;
571         (*q).parent() = s;
572         (*q).left() = d;
573         (*q).right() = b;
574         if (b != 0) {
575       (*d).parent() = q;
576       (*b).parent() = q;
577         }
578         (*s).height() += 1;
579         insert_mark(s);
580         remove_node(q);
581         if ((*s).is_root()) {
582       remove_mark(s);
583       remove_node(s);
584         }
585         assert(is_valid());
586         return s;
587       }
588
589       E* pair_transformation(E* q, E* s) {
590         //       std::cout << "pair: " << (*q).element() << " " << (*s).element() << "\n";
591         //       show();
592         assert(q != s);
593         assert((*q).height() == (*s).height());
594         assert(! (*q).is_root() && ! (*s).is_root());
595         E* p = (*q).parent();
596         assert(q == (*p).right());
597         E* r = (*s).parent();
598         assert(s == (*r).right());
599         remove_mark(q);
600         remove_mark(s);
601         bool x = comparator((*p).element(), (*r).element());
602         bool y = comparator((*q).element(), (*s).element());
603         int item = 2 * x + y;
604         switch (item) {
605         case 0 /* p >= r, q >= s */:
606       pair_action(r, p, s, q);
607       return q;
608         case 1 /* p >= r, q < s */:
609       pair_action(r, p, q, s);
610       return s;
611         case 2 /* p < r, q >= s */:
612       pair_action(p, r, s, q);
613       return q;
614         case 3 /* p < r, q < s */:
615       pair_action(p, r, q, s);
616       return s;
617         default:
```

```
618     assert(false);
619         }
620         return 0;
621       }
622
623       E* team_transformation() {
624         //       std::cout << "team:\n";
625         //       show();
626         height_type h = teams.choose();
627         index_type i = marks[h].choose();
628         E* q = nodes[i];
629         assert(is_marked(q));
630         marks[h].unset(i);
631         index_type j = marks[h].choose();
632         marks[h].set(i);
633         E* s = nodes[j];
634         assert(is_marked(s));
635         return pair_transformation(q, s);
636       }
637     };
638   }
639
640   #if defined(UNITTEST_SIMPLE_MARK_STORE)
641   #include "proxy-list-heap-store.h++"
642   #include "fat-weak-heap-node.h++"
643   #include <functional>
644   #include <memory>
645   #include "heap-proxy.h++"
646   #include "multiple-heap-framework.h++"
647
648   template <typename V, typename E, typename A>
649   E* create(V const& v, A& allocator) {
650     E* p = allocator.allocate(1);
651     new (p) E(v, allocator);
652     return p;
653   }
654
655   template <typename V, typename E, typename A>
656   void destroy(E* p, A& allocator) {
657     p->~E();
658     allocator.deallocate(p, 1);
659   }
660
661   template <typename V, typename C, typename A, typename E,
662       typename H, typename M>
663   void test_mark_store() {
664     typedef cphstl::multiple_heap_framework<V, C, A, E, H, M> Q;
665     Q queue;
666
667     typedef typename A::template rebind<E>::other node_allocator_type;
668     node_allocator_type node_allocator;
669     E* p = create<V, E, node_allocator_type>(V(2), node_allocator);
670     E* q = create<V, E, node_allocator_type>(V(4), node_allocator);
671     E* r = create<V, E, node_allocator_type>(V(1), node_allocator);
672     E* s = create<V, E, node_allocator_type>(V(7), node_allocator);
673     E* t = create<V, E, node_allocator_type>(V(11), node_allocator);
674     E* u = create<V, E, node_allocator_type>(V(3), node_allocator);
675     E* v = create<V, E, node_allocator_type>(V(5), node_allocator);
676     E* w = create<V, E, node_allocator_type>(V(8), node_allocator);
677     E* x = create<V, E, node_allocator_type>(V(9), node_allocator);
678     E* y = create<V, E, node_allocator_type>(V(6), node_allocator);
679     E* z = create<V, E, node_allocator_type>(V(10), node_allocator);
680
681     queue.insert(p);
682     queue.insert(q);
683     queue.insert(r);
684     queue.insert(s);
685     queue.insert(t);
686     queue.insert(u);
687     queue.insert(v);
688     queue.insert(w);
689     queue.insert(x);
690     queue.insert(y);
691     queue.insert(z);
```

```
692     assert(queue.size() == 11);
693     assert(queue.top() == t);
694 
695     H heap_store;
696     M mark_store;
697 
698     heap_store.inject(t, 3, mark_store);
699     heap_store.inject(x, 1, mark_store);
700     heap_store.inject(z, 0, mark_store);
701 
702     assert(heap_store.is_valid(mark_store));
703     assert(mark_store.is_valid());
704 
705     mark_store.mark(u);
706     mark_store.reduce(heap_store);
707     assert(heap_store.is_valid(mark_store));
708     assert(mark_store.is_valid());
709 
710     mark_store.mark(v);
711     mark_store.reduce(heap_store);
712     assert(heap_store.is_valid(mark_store));
713     assert(mark_store.is_valid());
714 
715     mark_store.mark(r);
716     mark_store.reduce(heap_store);
717     assert(heap_store.is_valid(mark_store));
718     assert(mark_store.is_valid());
719 
720     mark_store.mark(s);
721     mark_store.reduce(heap_store);
722     assert(heap_store.is_valid(mark_store));
723     assert(mark_store.is_valid());
724 
725     mark_store.mark(q);
726     mark_store.reduce(heap_store);
727     assert(heap_store.is_valid(mark_store));
728     assert(mark_store.is_valid());
729 
730     mark_store.mark(p);
731     mark_store.reduce(heap_store);
732     assert(heap_store.is_valid(mark_store));
733     assert(mark_store.is_valid());
734 
735     mark_store.mark(t);
736     mark_store.reduce(heap_store);
737     assert(heap_store.is_valid(mark_store));
738     assert(mark_store.is_valid());
739 
740     mark_store.mark(x);
741     mark_store.reduce(heap_store);
742     assert(heap_store.is_valid(mark_store));
743     assert(mark_store.is_valid());
744 
745     mark_store.mark(y);
746     mark_store.reduce(heap_store);
747     assert(heap_store.is_valid(mark_store));
748     assert(mark_store.is_valid());
749 
750     mark_store.mark(z);
751     mark_store.reduce(heap_store);
752     assert(heap_store.is_valid(mark_store));
753     assert(mark_store.is_valid());
754 
755     mark_store.unmark(z);
756     assert(heap_store.is_valid(mark_store));
757     assert(mark_store.is_valid());
758     mark_store.unmark(y);
759     assert(heap_store.is_valid(mark_store));
760     assert(mark_store.is_valid());
761     mark_store.unmark(x);
762     assert(heap_store.is_valid(mark_store));
763     assert(mark_store.is_valid());
764     mark_store.unmark(t);
765     assert(heap_store.is_valid(mark_store));
```

```
766     assert(mark_store.is_valid());
767     mark_store.unmark(p);
768     assert(heap_store.is_valid(mark_store));
769     assert(mark_store.is_valid());
770     mark_store.unmark(q);
771     assert(heap_store.is_valid(mark_store));
772     assert(mark_store.is_valid());
773     mark_store.unmark(s);
774     assert(heap_store.is_valid(mark_store));
775     assert(mark_store.is_valid());
776     mark_store.unmark(r);
777     assert(heap_store.is_valid(mark_store));
778     assert(mark_store.is_valid());
779     mark_store.unmark(v);
780     assert(heap_store.is_valid(mark_store));
781     assert(mark_store.is_valid());
782     mark_store.unmark(u);
783     assert(heap_store.is_valid(mark_store));
784     assert(mark_store.is_valid());
785
786     std::pair<E*, std::size_t> small = heap_store.eject();
787     assert(small.first == z);
788     assert(small.second == 0);
789
790     std::pair<E*, std::size_t> medium = heap_store.eject();
791     assert(medium.first == x);
792     assert(medium.second == 1);
793
794     std::pair<E*, std::size_t> big = heap_store.eject();
795     assert(big.first == t);
796     assert(big.second == 3);
797
798     destroy<V, E, node_allocator_type>(p, node_allocator);
799     destroy<V, E, node_allocator_type>(q, node_allocator);
800     destroy<V, E, node_allocator_type>(r, node_allocator);
801     destroy<V, E, node_allocator_type>(s, node_allocator);
802     destroy<V, E, node_allocator_type>(t, node_allocator);
803     destroy<V, E, node_allocator_type>(u, node_allocator);
804     destroy<V, E, node_allocator_type>(v, node_allocator);
805     destroy<V, E, node_allocator_type>(w, node_allocator);
806     destroy<V, E, node_allocator_type>(x, node_allocator);
807     destroy<V, E, node_allocator_type>(y, node_allocator);
808     destroy<V, E, node_allocator_type>(z, node_allocator);
809   }
810
811   int main(int, char**) {
812     typedef int V;
813     typedef std::less<V> C;
814     typedef std::allocator<V> A;
815     typedef cphstl::fat_weak_heap_node<V, A> E;
816     typedef cphstl::proxy_list_heap_store<C, A, E> H;
817     typedef cphstl::simple_mark_store<C, A, E> M;
818
819     test_mark_store<V, C, A, E, H, M>();
820   }
821
822   #endif
823   #endif
```

G.18 single-heap-framework.h++

```
1   /*
2     A priority-queue framework for a single array-based heap.
3
4     Authors: Stefan Edelkamp, Jyrki Katajainen Â© 2010
5   */
6
7   #ifndef _CPHSTL_SINGLE_HEAP_FRAMEWORK_
8   #define _CPHSTL_SINGLE_HEAP_FRAMEWORK_
9
10  #include <algorithm>
11  #include "allocator-proxy.h++"
12  #include "assert.h++"
```

```
13 #include "comparator-proxy.h++"
14 #include <cstddef>
15 #include <cstdlib>
16 #include "element-encapsulator.h++"
17 #include <functional>
18 #include <iostream>
19 #include <memory>
20 #include "top-down-binary-heap-heapifier.h++"
21 #include <vector>
22 #include "vector-surrogate.h++"
23 #include <utility>
24
25 namespace cphstl {
26
27   template <
28     typename V,
29     typename C = std::less<V>,
30     typename A = std::allocator<V>,
31     typename E = element_encapsulator<V, std::ptrdiff_t, A>,
32     typename H = top_down_binary_heap_heapifier,
33     typename K = std::vector<E*, A>,
34     typename S = vector_surrogate<E*, K>
35   >
36   class single_heap_framework {
37   public:
38
39     // types
40
41     typedef V value_type;
42     typedef C comparator_type;
43     typedef A allocator_type;
44     typedef E encapsulator_type;
45     typedef H heapifier_type;
46     typedef K kernel_type;
47     typedef S surrogate_type;
48     typedef std::size_t size_type;
49     typedef std::ptrdiff_t difference_type;
50     typedef V& reference;
51     typedef V const& const_reference;
52
53   protected:
54
55     typedef typename A::template rebind<S>::other surrogate_allocator_type;
56
57     comparator_proxy<C> comparator;
58     allocator_proxy<surrogate_allocator_type> surrogate_allocator;
59     K vector;
60     H heapifier;
61
62   public:
63
64     S* surrogate;
65
66     // structors
67
68     explicit single_heap_framework(C const& c = C(), A const& a = A(),
69           H const& h = H())
70       : comparator(c), surrogate_allocator(a), vector(a), heapifier(h) {
71       surrogate = surrogate_allocator.allocate(1);
72       (*surrogate).subject() = &vector;
73     }
74
75     ~single_heap_framework() {
76       surrogate_allocator.deallocate(surrogate, 1);
77     }
78
79     // iterators
80
81     E* begin() const {
82       if (vector.size() == 0) {
83   return (E*) 0;
84       }
85       return vector[0];
86     }
```

```
87
88      E* end() const {
89        return (E*) 0;
90      }
91
92      // accessors
93
94      A get_allocator() const {
95        return A(vector.get_allocator());
96      }
97
98      C get_comparator() const {
99        return comparator.subject();
100     }
101
102     size_type size() const {
103       return vector.size();
104     }
105
106     size_type max_size() const {
107       typename std::vector<int, A>::allocator_type a;
108       size_type available_memory = a.max_size() * sizeof(int); // in bytes
109       return available_memory / (sizeof(E) + 2 * sizeof(E*));
110     }
111
112     E* top() const {
113       return vector[0];
114     }
115
116     // modifiers
117
118     void insert(E* node) {
119       size_type last = vector.size();
120       (*node).position() = last;
121       vector.push_back(node);
122       heapifier.siftup(comparator, vector, last);
123     }
124
125     E* extract() {
126       assert(vector.size() != 0);
127       E* node = vector.back();
128       vector.pop_back();
129       return node;
130     }
131
132     void extract(E* node) {
133       if (vector.size() == 1) {
134   assert(vector[0] == node);
135   vector.pop_back();
136   return;
137       }
138       size_type i = std::abs(difference_type((*node).position()));
139       size_type last = vector.size() - 1;
140       std::swap(vector[i], vector[last]);
141       std::swap(vector[i]->position(), vector[last]->position());
142       size_type j = heapifier.siftdown(comparator, vector, last, i);
143       if (i == j ) {
144   heapifier.siftup(comparator, vector, j);
145       }
146       vector.pop_back();
147     }
148
149     void increase(E* node, V const& v) {
150       assert(! comparator(v, (*node).element()));
151       (*node).element() = v;
152       size_type i = std::abs(difference_type((*node).position()));
153       heapifier.siftup(comparator, vector, i);
154     }
155
156     void meld(single_heap_framework& other) {
157       if (size() > other.size()) {
158   swap(other);
159       }
160       while (other.vector.size() != 0) {
```

```
161     insert(other.extract());
162       }
163     }
164
165     void swap(single_heap_framework& other) {
166       std::swap(comparator, other.comparator);
167       std::swap(surrogate_allocator, other.surrogate_allocator);
168       std::swap(vector, other.vector);
169       std::swap(heapifier, other.heapifier);
170       std::swap(surrogate, other.surrogate);
171     }
172
173 #ifdef DEBUG
174
175     size_type parent(size_type j) {
176       return ((j - 1) / 2);
177     }
178
179     bool is_valid() {
180       return heapifier.is_valid(comparator, vector);
181     }
182
183     void show() {
184       size_type size = vector.size();
185       std::cout << std::endl << "size: " << size << std::endl;
186       std::cout << "values:" << std::endl;
187       for (size_type i = 0; i < size; ++i) {
188   std::cout << vector[i]->element() << " ";
189       }
190       std::cout << "\n";
191       std::cout << "indices:" << std::endl;
192       for (size_type i = 0; i < size; ++i) {
193   std::cout << vector[i]->position() << " ";
194       }
195       std::cout << "\n";
196       std::cout << "\n ----------------------------------- \n";
197     }
198
199 #endif
200
201   };
202 }
203
204 #endif
205 #ifdef UNITTEST_SINGLE_HEAP_FRAMEWORK
206
207 #include "weak-heap-heapifier.h++"
208
209 long comps = 0;
210
211 template <typename V>
212 class counting_comparator {
213 public:
214
215   bool operator()(V const& a, V const& b) const {
216     ++comps;
217     return a < b;
218   }
219 };
220
221 template <typename R>
222 void unittest_single_heap_framework() {
223   typedef typename R::encapsulator_type E;
224   typedef typename E::value_type V;
225   typedef std::size_t size_type;
226
227   R heap;
228
229   std::vector<E*> v;
230   size_type number_of_elements = 1000000;
231   comps = 0;
232   size_type total = 0;
233   std::cout<< "n: " << number_of_elements << std::endl;
234
```

```
235     for (size_type i = 0; i < number_of_elements; i++) {
236       E* p = new E(V(rand() % 100));
237       heap.insert(p);
238       v.push_back(p);
239       assert(heap.is_valid());
240     }
241     std::cout << "insert: comps per operation: "
242         << (double) comps / number_of_elements << std::endl;
243     total += comps;
244 
245     comps = 0;
246     for (size_type i = 0; i < number_of_elements; ++i) {
247       heap.increase(v[i], 100 + v[i]->element());
248       assert(heap.is_valid());
249     }
250     std::cout << "decrease: comps per operation: "
251         << (double) comps / number_of_elements << std::endl;
252     total += comps;
253 
254     comps = 0;
255     for (size_type i = 0; i < number_of_elements / 2; i++) {
256       heap.extract(v[i]);
257       delete v[i];
258       assert(heap.is_valid());
259     }
260     std::cout << "extract: comps per operation: "
261         << (double) comps / (number_of_elements / 2) << std::endl;
262     total += comps;
263 
264     comps = 0;
265     while (heap.size() > 0) {
266       E* p = heap.top();
267       heap.extract(p);
268       delete p;
269       assert(heap.is_valid());
270     }
271     std::cout << "pop: comps per operation: "
272         << (double) comps / (number_of_elements / 2) << std::endl;
273 
274     std::cout << "\ntotal number of comparisons: " << total << std::endl;
275   }
276 
277   int main() {
278     typedef double V;
279     typedef counting_comparator<V> C;
280     typedef cphstl::single_heap_framework<V, C> R;
281     unittest_single_heap_framework<R>();
282     typedef std::allocator<V> A;
283     typedef cphstl::element_encapsulator<V, std::ptrdiff_t , A> E;
284     typedef cphstl::weak_heap_heapifier H;
285     typedef cphstl::single_heap_framework<V, C, A, E, H> Rx;
286     unittest_single_heap_framework<Rx>();
287   }
288   #endif
```

G.19 top-down-binary-heap-heapifier.h++

```
1   /*
2     A top-down binary-heap heapifier works as proposed by Williams in
3     his seminal paper.
4 
5     Authors: Stefan Edelkamp, Jyrki Katajainen Â© 2009, 2010
6   */
7 
8   #ifndef _CPHSTL_TOP_DOWN_BINARY_HEAP_HEAPIFIER_
9   #define _CPHSTL_TOP_DOWN_BINARY_HEAP_HEAPIFIER_
10 
11  #include <algorithm> // std::swap
12 
13  namespace cphstl {
14 
15    class top_down_binary_heap_heapifier {
16    public:
```

```
17
18      template <typename C, typename K, typename I>
19      void siftup(C const& comparator, K& vector, I j) {
20        I i = parent(j);
21        while ((j != 0) && comparator(vector[i]->element(), vector[j]->element())) {
22    std::swap(vector[i], vector[j]);
23    std::swap(vector[i]->position(), vector[j]->position());
24    j = i;
25    i = parent(j);
26        }
27      }
28
29      template <typename C, typename K, typename I>
30      I siftdown(C const& comparator, K& vector, I n, I i) {
31        I j = first_child(i);
32        while (j < n) {
33    if ((j + 1 < n)
34        && comparator(vector[j]->element(), vector[j + 1]->element())) {
35      j += 1;
36    }
37    if (comparator(vector[i]->element(), vector[j]->element())) {
38      std::swap(vector[i], vector[j]);
39      std::swap(vector[i]->position(), vector[j]->position());
40      i = j;
41      j = first_child(i);
42    }
43    else return i;
44        }
45        return i;
46      }
47
48  #ifdef DEBUG
49
50      template <typename C, typename K>
51      bool is_valid(C const& comparator, K& vector) {
52        bool okey = true;
53        for (typename K::size_type i = 1; i < vector.size(); ++i) {
54    if (comparator(vector[parent(i)]->element(), vector[i]->element())) {
55      std::cout << "error: " << vector[parent(i)]->element() << "&"
56          << vector[i]->element() << std::endl;
57      okey = false;
58    }
59        }
60        return okey;
61      }
62
63  #endif
64
65    private:
66
67      template <typename I>
68      I parent(I j) {
69        return ((j - 1) / 2);
70      }
71
72      template <typename I>
73      I first_child(I j) {
74        return (2 * j + 1);
75      }
76    };
77  }
78
79  #endif
```

G.20 vector-encapsulator.h++

```
1   #ifndef _CPHSTL_VECTOR_SURROGATE_
2   #define _CPHSTL_VECTOR_SURROGATE_
3
4   /*
5     This surrogate keeps a pointer to a vector and accesses its elements
6     on behalf of it.
7
```

```
8     Author: Jyrki Katajainen Â© 2010
9   */
10
11  namespace cphstl {
12
13    template <typename V, typename S>
14    class vector_surrogate {
15    public:
16
17      typedef V slot_type;
18      typedef S subject_type;
19
20      subject_type*& subject() {
21        return ptr;
22      }
23
24      template <typename I>
25      slot_type& access(I index) {
26        return (*ptr)[index];
27
28      }
29
30    private:
31
32      subject_type* ptr;
33    };
34  }
35
36  #endif
```

G.21 weak-heap-heapifier.h++

```
1   /*
2     A weak-heap heapifier works as proposed by Edelkamp & Wegener in
3     their STACS paper. However, the bits are not stored as such, but a
4     negative index in the encapsulator denotes a 1-bit.
5
6     Authors: Stefan Edelkamp, Jyrki Katajainen Â© 2009, 2010
7   */
8
9   #ifndef _CPHSTL_WEAK_HEAP_HEAPIFIER_
10  #define _CPHSTL_WEAK_HEAP_HEAPIFIER_
11
12  #include <algorithm> // std::swap
13
14  namespace cphstl {
15
16    class weak_heap_heapifier {
17    public:
18
19      template <typename C, typename K, typename I>
20      void siftup(C const& comparator, K& vector, I j) {
21        if (j == 0) {
22    return;
23        }
24        while (j != 0) {
25    I i = distinguished_ancestor(j, vector);
26    if (comparator(vector[i]->element(), vector[j]->element())) {
27      I tmp_i = vector[i]->position();
28      I tmp_j = vector[j]->position();
29      vector[i]->position() = -tmp_j; // flip
30      vector[j]->position() = tmp_i;
31      std::swap(vector[i], vector[j]);
32      j = i;
33    }
34    else {
35      return;
36    }
37        }
38      }
39
40      template <typename C, typename K, typename I>
41      I siftdown(C const& comparator, K& vector, I n, I i) {
```

```
42          I k = second_child(i, vector);
43          if (k > n - 1) {
44        return i;
45          }
46          k = first_child(k, vector);
47          while (k < n) {
48        k = first_child(k, vector);
49          }
50          k = parent(k);
51          while (k > i) {
52        if (comparator(vector[i]->element(), vector[k]->element())) {
53          std::swap(vector[i], vector[k]);
54          I tmp = vector[i]->position();
55          vector[i]->position() = vector[k]->position();
56          vector[k]->position() = -tmp;
57        }
58        k = parent(k);
59          }
60          return i;
61        }
62
63      #ifdef DEBUG
64
65          template <typename C, typename K>
66          bool is_valid(C const& comparator, K& vector) {
67            bool validity = true;
68            for (typename K::size_type j = vector.size(); j > 1;) {
69        --j;
70        typename K::size_type i = distinguished_ancestor(j, vector);
71        if (comparator(vector[i]->element(), vector[j]->element())) {
72          std::cout << "half-order violation: " << j << " " << vector[j]->element() << "\n";
73          validity = false;
74        }
75            }
76            return validity;
77          }
78
79      #endif
80
81          //  private:
82
83          template <typename I>
84          I parent(I j) {
85            return j / 2;
86          }
87
88          template <typename I, typename K>
89          inline I distinguished_ancestor(I j, K& vector) {
90            assert(j != 0);
91            I i = parent(j);
92            while ((j & 1) == (vector[i]->position() < 0)) {
93        j = i;
94        i = parent(i);
95            }
96            return i;
97          }
98
99          template <typename I, typename K>
100         inline I first_child(I j, K& vector) {
101           return (2 * j + (vector[j]->position() < 0));
102         }
103
104         template <typename I, typename K>
105         inline I second_child(I j, K& vector) {
106           return (2 * j + 1 - (vector[j]->position() < 0));
107         }
108       };
109     }
110
111     #endif
```

G.22 weak-heap-node.h++

```
/*
  A weak-heap node

  Author: Jyrki Katajainen Â© 2009
*/

#ifndef _CPHSTL_WEAK_HEAP_NODE_
#define _CPHSTL_WEAK_HEAP_NODE_

#include "heap-node.h++"

namespace cphstl {

  template <typename V, typename A>
  class weak_heap_node
    : public heap_node<V, A, weak_heap_node<V, A> > {

  public:

    typedef V value_type;
    typedef A allocator_type;
    typedef weak_heap_node<V, A> N;

  private:

    weak_heap_node();
    weak_heap_node(N const&);
    weak_heap_node& operator = (N const&);

  public:

    weak_heap_node(V const& v, A const& a)
      : heap_node<V, A, N>(v, a) {
    }

#ifdef DEBUG

    template <typename C, typename M>
    bool is_valid(C const& comparator, M const& mark_store) const {
      N const* t = (*this).down_cast(this);
      bool valid = true;
      if ((*t).parent() != 0) {
        valid &= t -> parent() -> left() == t ||
          t -> parent() -> right() == t;
  if (! valid) std::cout << "parent\n";
      }
      if (! (*t).is_root() && (*t).left() != 0) {
        valid &= t -> left() -> parent() == t;
  if (! valid) std::cout << "left\n";
      }
      if ((*t).right() != 0) {
        valid &= t -> right() -> parent() == t;
  if (! valid) {
    std::cout << "right\n";
    std::cout << "t: " << (*t).element() << " ";
    std::cout << "right: " << t -> right() -> element() << " ";
    std::cout << "up again: " << t -> right() -> parent() -> element() << "\n";
  }
      }
      if (! (*t).is_root() && ! mark_store.is_marked(t)) {
        valid &= ! comparator((*t).distinguished_ancestor() -> element(), (*t).element());

  if (! valid) std::cout << "ancestor\n";}
      return valid;
    }

#endif

  };
}

#endif
```

G.23 weak-queue-node.h++

```
/*
  A heap node used as a base for various specialized heap nodes

  Authors: Asger Bruun, Jyrki Katajainen Â© 2009, 2010
*/

#ifndef _CPHSTL_HEAP_NODE_
#define _CPHSTL_HEAP_NODE_

#include "assert.h++"
#include <cstddef> // std::size_t
#include <iostream>
#include <list>

namespace cphstl {

  template <typename V, typename A, typename N>
  class heap_node {
  public:

    typedef V value_type;
    typedef A allocator_type;
    typedef std::size_t size_type;
    typedef unsigned char height_type;
    typedef heap_node<V, A, N> self_type;

    struct hole_type {
      N* parent;
      N* current;
      union {
  N* left;
  void* owner;
      };

      hole_type(N* p)
        : parent((*p).parent()), current(p) {
  if (parent != 0) {
    left = (*p).left();
  }
  else {
    owner = (*p).owner();
  }
      }
    };

    N* parent_;
    union {
      N* left_;
      void* owner_;
    };
    N* right_;
    V value_;

  private:

    heap_node();
    heap_node(heap_node const&);
    heap_node& operator = (heap_node const&);

  protected:

    N const* const down_cast(self_type const* const b) const {
      return static_cast<N const* const>(b);
    }

    N* const down_cast(self_type* const b) const {
      return static_cast<N* const>(b);
    }

    N* const down_cast(self_type* const b) {
      return static_cast<N* const>(b);
```

```
72      }
73
74    public:
75
76      heap_node(V const& v, A const&)
77        : parent_(down_cast(0)), left_(down_cast(0)), right_(down_cast(0)),
78        value_(v) {
79      }
80
81      static size_type footprint() {
82        return sizeof(N);
83      }
84
85      bool is_root() const {
86        return parent_ == 0;
87      }
88
89      bool is_leaf() const {
90        return right_ == 0;
91      }
92
93      V const& element() const {
94        return value_;
95      }
96
97      V& element() {
98        return value_;
99      }
100
101     N* left() const {
102       return left_;
103     }
104
105     N*& left() {
106       return left_;
107     }
108
109     N* right() const {
110       return right_;
111     }
112
113     N*& right() {
114       return right_;
115     }
116
117     N* parent() const {
118       return parent_;
119     }
120
121     N*& parent() {
122       return parent_;
123     }
124
125     void* owner() const {
126       return owner_;
127     }
128
129     void*& owner() {
130       return owner_;
131     }
132
133     template <typename C, typename M>
134     N* join(N* q, C const& comparator, M&) {
135       N* p = down_cast(this);
136       if (comparator((*p).element(), (*q).element())) {
137         N* c = (*q).right();
138         if (c != 0) {
139           (*c).parent() = p;
140         }
141         (*p).left() = c;
142         (*q).right() = p;
143         (*p).parent() = q;
144         return q;
145       }
```

```
146         else {
147           N* c = (*p).right();
148           if (c != 0) {
149             (*c).parent() = q;
150           }
151           (*q).left() = c;
152           (*p).right() = q;
153           (*q).parent() = p;
154           return p;
155         }
156       }
157
158       template <typename C, typename M>
159       N* fast_join(N* q, N*, C const& comparator, M& mark_store) {
160         N* p = down_cast(this);
161         return (*p).join(q, comparator, mark_store);
162       }
163
164       template <typename C>
165       N* split(C const&) {
166         N* p = down_cast(this);
167         assert(p != 0);
168         assert((*p).right() != 0);
169         N* q = (*p).right();
170         N* r = (*q).left();
171         (*p).right() = r;
172         if (r != 0) {
173           (*r).parent() = p;
174         }
175         (*q).parent() = 0;
176         (*q).left() = 0;
177         return q;
178       }
179
180       void swap_roots(N* q) {
181         N* p = down_cast(this);
182         assert((*p).is_root());
183         assert((*q).is_root());
184         N* c = (*p).right();
185         N* g = (*q).right();
186         (*p).right() = g;
187         (*q).right() = c;
188         if (c != 0) {
189           (*c).parent() = q;
190         }
191         if (g != 0) {
192           (*g).parent() = p;
193         }
194       }
195
196       N* release_root() {
197         N* p = down_cast(this);
198         N* q = (*p). right();
199         (*p).right() = 0;
200         (*q).parent() = 0;
201         return q;
202       }
203
204       N* release_subheap() {
205         N* p = down_cast(this);
206         N* q = (*p).left();
207         (*p).left() = 0;
208         if (q != 0) {
209       (*q).parent() = 0;
210         }
211         return q;
212       }
213
214       hole_type splice_out() {
215         // Note: This function leaves the underlying tree broken,
216         // untraversable, and unprintable until splice_in.
217         assert(this != 0);
218         hole_type hole(down_cast(this));
219         (*this).parent() = 0;
```

```
220         (*this).left() = 0;
221         return hole;
222       }
223
224       template <typename H>
225       void splice_in(hole_type& hole, H& heap_store) {
226         assert(hole.current != 0);
227         N* p = down_cast(this);
228         (*p).parent() = hole.parent;
229         if (hole.parent != 0 && (*hole.parent).left() == hole.current) {
230           (*hole.parent).left() = p;
231         }
232         if (hole.parent != 0 && (*hole.parent).right() == hole.current) {
233           (*hole.parent).right() = p;
234         }
235         if (hole.parent == 0) {
236     typedef typename H::heap_proxy_type heap_proxy_type;
237     (*p).owner() = hole.owner;
238     heap_store.replace(p);
239         }
240         else {
241     (*this).left() = hole.left;
242     if (hole.left != 0) {
243       (*hole.left).parent() = p;
244     }
245         }
246       }
247
248   #ifdef FIRST_VERSION
249
250       void swap_nodes(N* q) {
251         // Warning: The backpointers at owners are not corrected because
252         // the type of the owners is not known.
253         N* p = down_cast(this);
254         assert(p != 0);
255         assert(q != 0);
256         N* a = (*p).parent_;
257         N* b = (*p).left_;
258         N* c = (*p).right_;
259         N* e = (*q).parent_;
260         N* f = (*q).left_;
261         N* g = (*q).right_;
262
263         if (a != 0 && (*a).left_ == p) {
264           (*a).left_ = q;
265         }
266         if (a != 0 && (*a).right_ == p) {
267           (*a).right_ = q;
268         }
269         if (b != 0 && (! (*p).is_root()) && (*b).parent_ == p) {
270           (*b).parent_ = q;
271         }
272         if (c != 0 && (*c).parent_ == p) {
273           (*c).parent_ = q;
274         }
275         if (e != 0 && (*e).left_ == q) {
276           (*e).left_ = p;
277         }
278         if (e != 0 && (*e).right_ == q) {
279           (*e).right_ = p;
280         }
281         if (f != 0 && (! (*p).is_root()) && (*f).parent_ == q) {
282           (*f).parent_ = p;
283         }
284         if (g != 0 && (*g).parent_ == q) {
285           (*g).parent_ = p;
286         }
287
288         (*p).parent_ = e;
289         (*p).left_ = f;
290         (*p).right_ = g;
291         (*q).parent_ = a;
292         (*q).left_ = b;
293         (*q).right_ = c;
```

```
294
295         if (a == q && f == p) {
296           (*p).left_ = q;
297           (*q).parent_ = p;
298           return;
299         }
300         if (a == q && g == p) {
301           (*p).right_ = q;
302           (*q).parent_ = p;
303           return;
304         }
305         if (b == q) {
306           (*p).parent_ = q;
307           (*q).left_ = p;
308           return;
309         }
310         if (c == q) {
311           (*p).parent_ = q;
312           (*q).right_ = p;
313           return;
314         }
315       }
316
317   #endif
318
319       template <typename C>
320       N* distinguished_descendant(C const&) const {
321         N const* q = down_cast(this);
322         assert(q != 0);
323         return ((*q).is_root())? (*q).right() : (*q).left();
324       }
325
326       N* distinguished_ancestor() const {
327         N const* q = down_cast(this);
328         assert(q != 0);
329         N* p = (*q).parent();
330         while (p != 0 && (*p).left() == q) {
331           q = p;
332           p = (*p).parent();
333         }
334         return p;
335       }
336
337       N* promote(N* p) {
338         N* q = down_cast(this);
339         assert(p == (*q).distinguished_ancestor());
340         if (p == (*q).parent()) {
341           assert((*p).right() == q);
342           N* a = (*p).parent();
343           N* b = (*p).left();
344           N* f = (*q).left();
345           N* g = (*q).right();
346           (*p).parent() = q;
347           (*p).left() = f;
348           (*p).right() = g;
349           (*q).parent() = a;
350           (*q).left() = b;
351           (*q).right() = p;
352           if (a != 0) {
353             if ((*a).right() == p) {
354               (*a).right() = q;
355             }
356             else {
357               if (! (*p).is_root()) {
358                 (*a).left() = q;
359               }
360             }
361           }
362           if (b != 0 && (! (*p).is_root())) {
363             (*b).parent() = q;
364           }
365           if (f != 0) {
366             (*f).parent() = p;
367           }
```

```
368           if (g != 0) {
369             (*g).parent() = p;
370           }
371         }
372         else {
373           N* a = (*p).parent();
374           N* b = (*p).left();
375           N* c = (*p).right();
376           N* e = (*q).parent();
377           N* f = (*q).left();
378           N* g = (*q).right();
379           (*p).parent() = e;
380           (*p).left() = f;
381           (*p).right() = g;
382           (*q).parent() = a;
383           (*q).left() = b;
384           (*q).right() = c;
385           if (a != 0) {
386             if ((*a).right() == p) {
387               (*a).right() = q;
388             }
389             else {
390               if (! (*p).is_root()) {
391                 (*a).left() = q;
392               }
393             }
394           }
395           if (b != 0 && (! (*p).is_root())) {
396             (*b).parent() = q;
397           }
398           if (c != 0) {
399             (*c).parent() = q;
400           }
401           if (e != 0 && (*e).left() == q) {
402             (*e).left() = p;
403           }
404           if (e != 0 && (*e).right() == q) {
405             (*e).right() = p;
406           }
407           if (f != 0) {
408             (*f).parent_ = p;
409           }
410           if (g != 0) {
411             (*g).parent_ = p;
412           }
413         }
414         return q;
415       }
416
417       N const* root() const {
418         N const* p = down_cast(this);
419         assert(p != 0);
420         while (! (*p).is_root()) {
421           p = (*p).parent();
422         }
423         return p;
424       }
425
426       N const* successor() const {
427         N const* x = down_cast(this);
428         assert(x != 0);
429         if ((*x).right() != 0) {
430           x = (*x).right();
431           while ((*x).left() != 0) {
432             x = (*x).left();
433           }
434           return x;
435         }
436         N const* y = (*x).parent();
437         while (y != 0 && x == (*y).right()) {
438           x = y;
439           y = (*y).parent();
440         }
441         return y;
```

```
442       }
443
444       height_type height() const {
445         N const* p = down_cast(this);
446         assert(p != 0);
447         height_type h = 0;
448         while (! (*p).is_leaf()) {
449           p = (*p).right();
450     h += 1;
451         }
452         return h;
453       }
454
455   #ifdef DEBUG
456
457       void show_tree() const {
458         N const* t = down_cast(this);
459         std::cout << (*t).element() << "␣";
460         std::cout << "\n";
461         std::cout.flush();
462         t = (*t).right();
463         if (t == 0) {
464     return;
465         }
466         std::list<N const*> level;
467         level.push_front(t);
468         while (! level.empty()) {
469           typename std::list<N const*>::iterator last = level.end();
470           --last;
471           typename std::list<N const*>::iterator p = level.begin();
472           bool stop = false;
473           while (! stop) {
474             N const* t = *p;
475             if (p == last) stop = true;
476             ++p;
477             level.pop_front();
478             if ((*t).right() != 0) {
479               level.push_back((*t).left());
480               level.push_back((*t).right());
481             }
482             std::cout << (*t).element() << "␣";
483           }
484           std::cout << "\n";
485           std::cout.flush();
486         }
487       }
488
489   #endif
490
491     };
492   }
493
494   #endif
```

H CPHSTL_Branch/Type/Code

H.1 type.h++

```
/*

  Author: Jyrki Katajainen 2001, 2006, 2009


  This module is a CPH STL extension which provides a collection of
  tools for performing type mappings. The purpose of this module is to
  make the library self-contained, but these tools can be useful in
  other contexts as well.

  The main sources when writing this have been

  Andrei Alexandrescu. Modern C++ Design: Generic Programming and
  Design Patterns Applied, Addison-Wesley (2001), see Chapters 2 and 3.

  Jaakko Järvi, Tuple types and multiple return values, C/C++ Users
  Journal 19,8 (2001), 24-35.

  David Vandevoorde and Nicolai M. Josuttis. C++ Templates: The
  Complete Guide. Addison-Wesley (2003), see Sections 13.10 and 15.2
  and Chapter 19.

  Note: Use <tr1/type_traits> if possible.
*/

#ifndef __CPHSTL_TYPE__
#define __CPHSTL_TYPE__

namespace cphstl {

  template <int v>
  class int2type;

  template <bool V>
  class bool2type;

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

}

#include "type.i++" // Implements the type functions

namespace cphstl {

  template <typename T, typename U>
  class types {
  public:
    enum {
      are_same = cphstl::same_traits<T, U>::positive,
      are_not_same = cphstl::same_traits<T, U>::negative
    };
  };

  template <typename T>
  class type {
  public:
    typedef T original_type;
    typedef typename non_volatile<typename non_const<T>::type>::type unqualified;
    typedef const typename cphstl::non_const<T>::type const_qualified;
    typedef volatile typename cphstl::non_volatile<T>::type volatile_qualified;
    typedef const volatile typename cphstl::type<T>::unqualified const_volatile_qualified;
    typedef typename cphstl::non_const<T>::type non_const;
    typedef typename non_volatile<T>::type non_volatile;
    typedef typename cphstl::reference_traits<T>::type reference;
    typedef typename cphstl::const_reference_traits<T>::type const_reference;
    typedef T* pointer;
    typedef T* const const_pointer;
```

```
69
70      enum {
71        is_unsigned_integral = cphstl::lookup_traits<T>::unsigned_integral,
72        is_signed_integral = cphstl::lookup_traits<T>::signed_integral,
73        is_other_integral = cphstl::lookup_traits<T>::other_integral,
74        is_integral = is_unsigned_integral || is_signed_integral ||
75                      is_other_integral,
76        is_floating_point = cphstl::lookup_traits<T>::floating_point,
77        is_arithmetic = is_integral || is_floating_point,
78        is_fundamental = is_arithmetic || is_floating_point ||
79                         cphstl::types<T, void>::are_same,
80        is_pointer = cphstl::pointer_traits<T>::positive,
81        is_reference = cphstl::reference_traits<T>::positive,
82        is_array = cphstl::array_traits<T>::positive,
83        is_function = cphstl::function_traits<T>::positive,
84        is_pointer_to_member = cphstl::pointer_to_member_traits<T>::positive,
85        //       is_enumeration = ?,
86        //       is_union = ?,
87        //       is_bit_copiable = ?,
88        is_class = cphstl::class_traits<T>::positive,
89        has_member_const_iterator = cphstl::member_const_iterator<T>::positive,
90        is_allocator = cphstl::allocator_traits<T>::positive
91      };
92    };
93
94    /* To be included:
95
96    Author: efim.slobodov233432@arcor.de, April 6th, 2005
97    Source: http://www.codecomments.com/archive324-2005-4-448963.html
98
99  typedef char Small;
100 typedef struct {char unused[2];} Big;
101
102 template<int>
103 class Selector;
104
105 template <typename T>
106 struct Has_member_dual {
107 private:
108   template <typename U>
109   static ::Small test(::Selector<sizeof(&U::dual)>*);
110
111   template <typename U>
112   static ::Big test(...);
113
114 public:
115   enum{value=sizeof(test<T>(0))==sizeof(::Small)};
116 };
117
118 // test
119
120 class my1 {
121 };
122
123 class my2 {
124 public:
125   void dual();
126 };
127
128 #include <iostream>
129
130 int main(int, char **) {
131   std::cout << "my1=" << Has_member_dual<my1>::value << "== 0?"
132             << std::endl;
133   std::cout << "my2=" << Has_member_dual<my2>::value << "== 1?"
134             << std::endl;
135   std::cout << "int=" << Has_member_dual<int>::value << "== 0?"
136             << std::endl;
137   return 0;
138 }
139 */
140 }
141
142 #endif // __CPHSTL_TYPE__
```

H.2 type.i++

```
/*
  Author: Jyrki Katajainen 2001, 2006
*/

#include <cstddef> // defines std::size_t
#include <memory> // defines std::allocator

namespace cphstl {

  // int to type conversion used for dispatching

  template <int v>
  class int2type {
  public:
    enum { value = v };
  };

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


  // if statement for meta-programming

  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 same_traits {
  public:
    enum { positive = 0 };
    enum { negative = 1 };
  };

  // An allocator? A correct test should check whether specific
  // typedefs and member functions exist!

  template <typename U>
  class allocator_traits {
  public:
    enum { positive = 0};
  };

  template <typename V>
  class allocator_traits <std::allocator<V> > {
  public:
    enum { positive = 1};
  };

  template <typename X>
  class same_traits <X, X> {
  public:
    enum { positive = 1 };
    enum { negative = 0 };
  };

  template <typename U>
  class non_const {
  public:
    typedef U type;
  };
```

```
73
74    template <typename U>
75    class non_const <U const> {
76    public:
77      typedef U type;
78    };
79
80    template <typename U>
81    class non_volatile {
82    public:
83      typedef U type;
84    };
85
86    template <typename U>
87    class non_volatile <volatile U> {
88    public:
89      typedef U type;
90    };
91
92    template <typename U>
93    class reference_traits {
94    public:
95      typedef U& type;
96      enum { positive = 0};
97    };
98
99    template <typename U>
100   class reference_traits <U&> {
101   public:
102     typedef U& type;
103     enum { positive = 1};
104   };
105
106   template <>
107   class reference_traits <void> {
108   public:
109     typedef void type;
110     enum {positive = 0};
111   };
112
113   template <typename U>
114   class const_reference_traits {
115   public:
116     typedef U const& type;
117   };
118
119   template <typename U>
120   class const_reference_traits <U const> {
121   public:
122     typedef U const& type;
123   };
124
125   template <typename U>
126   class const_reference_traits <U&> {
127   public:
128     typedef U const& type;
129   };
130
131   template <>
132   class const_reference_traits <void> {
133   public:
134     typedef void type;
135   };
136
137   template <typename U>
138   class pointer_traits {
139   public:
140     enum { positive = 0 };
141   };
142
143   template <typename U>
144   class pointer_traits <U*> {
145   public:
146     enum { positive = 1 };
```

```
147     };
148
149     template <typename T>
150     class lookup_traits {
151     public:
152       enum {
153         unsigned_integral = 0,
154         signed_integral = 0,
155         other_integral = 0,
156         floating_point = 0
157       };
158     };
159
160     template <>
161     class lookup_traits <bool> {
162     public:
163       enum {
164         unsigned_integral = 0,
165         signed_integral = 0,
166         other_integral = 1,
167         floating_point = 0
168       };
169     };
170
171     template <>
172     class lookup_traits <char> {
173     public:
174       enum {
175         unsigned_integral = 0,
176         signed_integral = 0,
177         other_integral = 1,
178         floating_point = 0
179       };
180     };
181
182     template <>
183     class lookup_traits <signed char> {
184     public:
185       enum {
186         unsigned_integral = 0,
187         signed_integral = 1,
188         other_integral = 0,
189         floating_point = 0
190       };
191     };
192
193     template <>
194     class lookup_traits <unsigned char> {
195     public:
196       enum {
197         unsigned_integral = 1,
198         signed_integral = 0,
199         other_integral = 0,
200         floating_point = 0
201       };
202     };
203
204     template <>
205     class lookup_traits <wchar_t> {
206     public:
207       enum {
208         unsigned_integral = 0,
209         signed_integral = 0,
210         other_integral = 1,
211         floating_point = 0
212       };
213     };
214
215     template <>
216     class lookup_traits <signed short> {
217     public:
218       enum {
219         unsigned_integral = 0,
220         signed_integral = 1,
```

```
221         other_integral = 0,
222         floating_point = 0
223       };
224     };
225
226     template <>
227     class lookup_traits <unsigned short> {
228     public:
229       enum {
230         unsigned_integral = 1,
231         signed_integral = 0,
232         other_integral = 0,
233         floating_point = 0
234       };
235     };
236
237     template <>
238     class lookup_traits <signed int> {
239     public:
240       enum {
241         unsigned_integral = 0,
242         signed_integral = 1,
243         other_integral = 0,
244         floating_point = 0
245       };
246     };
247
248     template <>
249     class lookup_traits <unsigned int> {
250     public:
251       enum {
252         unsigned_integral = 1,
253         signed_integral = 0,
254         other_integral = 0,
255         floating_point = 0
256       };
257     };
258
259     template <>
260     class lookup_traits <signed long> {
261     public:
262       enum {
263         unsigned_integral = 0,
264         signed_integral = 1,
265         other_integral = 0,
266         floating_point = 0
267       };
268     };
269
270     template <>
271     class lookup_traits <unsigned long> {
272     public:
273       enum {
274         unsigned_integral = 1,
275         signed_integral = 0,
276         other_integral = 0,
277         floating_point = 0
278       };
279     };
280
281   #if LONGLONG_EXISTS
282     template <>
283     class lookup_traits <signed long long> {
284     public:
285       enum {
286         unsigned_integral = 0,
287         signed_integral = 1,
288         other_integral = 0,
289         floating_point = 0
290       };
291     };
292
293     template <>
294     class lookup_traits <unsigned long long> {
```

```
295     public:
296       enum {
297         unsigned_integral = 1,
298         signed_integral = 0,
299         other_integral = 0,
300         floating_point = 0
301       };
302     };
303   #endif
304
305     template <>
306     class lookup_traits <float> {
307     public:
308       enum {
309         unsigned_integral = 0,
310         signed_integral = 0,
311         other_integral = 0,
312         floating_point = 1
313       };
314     };
315
316     template <>
317     class lookup_traits <double> {
318     public:
319       enum {
320         unsigned_integral = 0,
321         signed_integral = 0,
322         other_integral = 0,
323         floating_point = 1
324       };
325     };
326
327     template <>
328     class lookup_traits <long double> {
329     public:
330       enum {
331         unsigned_integral = 0,
332         signed_integral = 0,
333         other_integral = 0,
334         floating_point = 1
335       };
336     };
337
338     template <typename T>
339     class array_traits {
340     public:
341       enum { positive = 0};
342     };
343
344     template <typename T, std::size_t N>
345     class array_traits <T[N]> {
346     public:
347       enum { positive = 1};
348     };
349
350     template <typename T>
351     class array_traits <T[]> {
352     public:
353       enum { positive = 1};
354     };
355
356     template <typename T>
357     class function_traits {
358     private:
359       typedef char one;
360       typedef struct {char a[2]; } two;
361       template <typename U>
362       static one test(...);
363       template <typename U>
364       static two test(U (*)[1]);
365     public:
366       enum { positive = sizeof(function_traits<T>::template test<T>(0)) == 1 };
367     };
368
```

```
369     template <typename T>
370     class function_traits <T&> {
371     public:
372       enum { positive = 0 };
373     };
374
375     template <>
376     class function_traits <void> {
377     public:
378       enum { positive = 0 };
379     };
380
381     template <>
382     class function_traits <void const> {
383     public:
384       enum { positive = 0 };
385     };
386
387     template <>
388     class function_traits <void volatile> {
389     public:
390       enum { positive = 0 };
391     };
392
393     template <>
394     class function_traits <void const volatile> {
395     public:
396       enum { positive = 0 };
397     };
398
399     template <typename T>
400     class pointer_to_member_traits {
401     public:
402       enum { positive = 0 };
403     };
404
405     template <typename T, typename C>
406     class pointer_to_member_traits <T C::*> {
407     public:
408       enum { positive = 1 };
409     };
410
411     template <typename T>
412     class class_traits {
413     private:
414       typedef char one;
415       typedef struct {char a[2]; } two;
416       template <typename U>
417       static one test(int U::*);
418       template <typename U>
419       static two test(...);
420     public:
421       enum { positive = sizeof(class_traits<T>::template test<T>(0)) == 1 };
422     };
423
424     template <typename T>
425     class member_const_iterator {
426     private:
427       typedef char one;
428       typedef struct {char a[2]; } two;
429       template <typename U>
430       static one test(typename U::const_iterator const*);
431       template <typename U>
432       static two test(...);
433     public:
434       enum { positive = sizeof(test<T>(0)) == 1 };
435     };
436   }
```