

Weak-heap and weak-queue frameworks: Source code

Stefan Edelkamp¹, Amr Elmasry², and Jyrki Katajainen²

¹ *TZI, Universität Bremen*

Am Fallturm 1, 28357 Bremen, Germany

² *Department of Computer Science, University of Copenhagen
Universitetsparken 1, 2100 Copenhagen East, Denmark*

Abstract. This report is an electronic appendix to our paper “The weak-heap family of priority queues in theory and praxis” presented at the 18th Computing: Australasian Theory Symposium that was held in Melbourne in January-February 2012. This report together with an accompanying `tar` ball gives the source code used in the experiments reported in that paper.

We implemented weak heaps and their close relatives weak queues as component frameworks that utilize the same collection of registries. By varying the registries used, our frameworks provide implementations for six different data structures: weak heap, weak queue, rank-relaxed weak heap, rank-relaxed weak queue, run-relaxed weak heap, and run-relaxed weak queue. Frameworks made policy-based benchmarking possible. We carried out extensive benchmarking with Dijkstra’s algorithm to compute shortest-path distances, and some syntactic data sets exploring the performance of these data structures in a worst-case scenario.

In its standard form a weak heap is implemented using an array. For a weak heap of size n , *insert*, *decrease*, and *delete-min* operations all take $O(\lg n)$ time in the worst case. By relying on a pointer-based implementation and by allowing a logarithmic number of nodes to break weak-heap ordering, *insert* and *decrease* operations can be supported in $O(1)$ worst-case time. For any sequence of *insert*, *decrease*, and *delete-min* operations, the bound obtainable for the number of element comparisons performed by a rank-relaxed weak heap is the best for all known priority queues; the bound is even better than that known for a Fibonacci heap.

In our experiments, for random data sets, non-relaxed versions performed best and rank-relaxed versions were slightly faster than run-relaxed versions. Compared to weak-heap variants, the corresponding weak-queue variants were slightly better in time but not in the number of element comparisons. For worst-case data sets, non-relaxed versions were slow for *decrease* and weak-queue variants were slightly faster than weak-heap variants. For *insert*, the logarithmic worst-case was hardly visible for weak heaps, whereas for relaxed versions the overhead caused by complicated transformations was clearly visible. For *delete-min*, all six data structures performed well. As a conclusion, we had to admit that the progress made by us was mainly analytical.

Keywords. Priority queues, shortest paths, weak heaps, weak queues, relaxed weak heaps, relaxed weak queues.

Copyright notice

Copyright © 2000–2012 by The authors and Performance Engineering Laboratory (University of Copenhagen)

The programs included in the CPH STL are placed in the public domain. The files may be freely copied and distributed, provided that no changes whatsoever are made. Changes are permissible only if the modified files are given new names, different from the names of existing files in the CPH STL, and only if the modified files are clearly identified as not being part of the library. Usage of the source code in derived works is otherwise unrestricted.

The authors have tried to produce correct and useful programs, but no warranty of any kind should be assumed.

Release date

2012-05-14

Included files

File	Page
§ 1 meldable-priority-queue.h++	4
§ 2 meldable-priority-queue.i++	6
§ 3 priority-queue-iterator.h++	10
§ 4 priority-queue-iterator.i++	12
§ 5 comparator-proxy.h++	14
§ 6 relaxed-heap-modifier.h++	15
§ 7 naive-mark-registry.h++	26
§ 8 eager-mark-registry.h++	28
§ 9 lazy-mark-registry.h++	32
§ 10 heap-node.h++	38
§ 11 root-registry.h++	40
§ 12 relaxed-weak-queue.h++	43
§ 13 weak-queue.i++	47
§ 14 perfect-weak-heap-node.h++	48
§ 15 rank-relaxed-weak-queue.i++	49
§ 16 run-relaxed-weak-queue.i++	50
§ 17 level-registry.h++	51
§ 18 weak-heap.i++	54
§ 19 weak-heap-node.h++	54
§ 20 pointer-based-weak-heap.h++	57
§ 21 rank-relaxed-weak-heap.i++	60
§ 22 run-relaxed-weak-heap.i++	61
§ 23 graph-vertex.h++	62
§ 24 combined-node.h++	63
§ 25 graph-edge.h++	66
§ 26 directed-graph.h++	68
§ 27 dijkstra.c++	69
§ 28 push-time.c++	73
§ 29 push-comp.c++	74
§ 30 increase-time.c++	75
§ 31 increase-comp.c++	77
§ 32 pop-time.c++	77
§ 33 pop-comp.c++	78
§ 34 benchmark.mk	79

STL interface

§ 1 *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 §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 #ifndef __CPHSTL_MELDABLE_PRIORITY_QUEUE__
35 #define __CPHSTL_MELDABLE_PRIORITY_QUEUE__
36
37 #include <cstddef> // std::size_t and std::ptrdiff_t
38
39 namespace cphstl {
40     template <typename V, typename C, typename A, typename E, typename R,
41              typename I, typename J>
42     class meldable_priority_queue {
43     public:
44
45         // types
46
47         typedef V value_type;
48         typedef C comparator_type;
49         typedef A allocator_type;
50         typedef E encapsulator_type;
51         typedef R realizator_type;
52         typedef I iterator;
53         typedef J const_iterator;
54         typedef typename R::reference reference;
55         typedef typename R::const_reference const_reference;
56         typedef V* pointer;
57         typedef V const* const_pointer;
58         typedef std::size_t size_type;
59         typedef std::ptrdiff_t difference_type;
60         typedef std::reverse_iterator<iterator> reverse_iterator;

```

```

61     typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
62     typedef meldable_priority_queue<V, C, A, E, R, I, J> container_type;
63
64 protected:
65
66     typedef typename A::template rebind<E>::other encapsulator_allocator_type;
67
68     R realizator;
69     encapsulator_allocator_type allocator;
70
71     E* create(V const &);
72     void destroy(E*);
73
74     template <typename K>
75     void insert(K, K);
76
77 public:
78
79     // structs
80
81     explicit meldable_priority_queue(C const & = C(), A const & = A());
82     meldable_priority_queue(meldable_priority_queue const &);
83     meldable_priority_queue & operator=(meldable_priority_queue const &);
84     ~meldable_priority_queue();
85
86     // iterators
87
88     iterator begin();
89     const_iterator begin() const;
90     iterator end();
91     const_iterator end() const;
92
93     // accessors
94
95     allocator_type get_allocator() const;
96     comparator_type get_comparator() const;
97     bool empty() const;
98     size_type size() const;
99     size_type max_size() const;
100    const_iterator top() const;
101
102    // modifiers
103
104    iterator top();
105    iterator push(V const &);
106    void pop();
107    void erase(iterator);
108    void increase(iterator, V const &);
109    void clear();
110    void meld(meldable_priority_queue &);
111    void swap(meldable_priority_queue &);
112
113 };
114
115 // algorithms
116
117 template <typename V, typename C, typename A, typename E,
118         typename R, typename I, typename J>
119 meldable_priority_queue<V, C, A, E, R, I, J>&
120 meld(meldable_priority_queue<V, C, A, E, R, I, J>&,
121     meldable_priority_queue<V, C, A, E, R, I, J>&);
122
123 template <typename V, typename C, typename A, typename E,
124         typename R, typename I, typename J>
125 void swap(meldable_priority_queue<V, C, A, E, R, I, J>&,

```

```

126         meldable_priority_queue<V, C, A, E, R, I, J>&);
127     }
128
129     #include "meldable-priority-queue.i++"
130 #endif

```

§ 2 *meldable-priority-queue.i++*

```

1  /*
2  2  A meldable priority queue is a container class that just calls the
3  3  functions available in the realizator class.
4  4
5  5  Author: Jyrki Katajainen © 2006, 2009, 2010
6  6  */
7  7
8  8  namespace cphstl {
9  9  template <typename V, typename C, typename A, typename E,
10 10         typename R, typename I, typename J>
11 11  E*
12 12  meldable_priority_queue<V, C, A, E, R, I, J>::create(V const & v) {
13 13      E* p = allocator.allocate(1);
14 14      try {
15 15          new (p) E(v, allocator);
16 16      }
17 17      catch (...) {
18 18          destroy(p);
19 19          throw;
20 20      }
21 21      return p;
22 22  }
23 23
24 24  template <typename V, typename C, typename A, typename E,
25 25         typename R, typename I, typename J>
26 26  void
27 27  meldable_priority_queue<V, C, A, E, R, I, J>::destroy(E* p) {
28 28      p->~E();
29 29      allocator.deallocate(p, 1);
30 30  }
31 31
32 32  template <typename V, typename C, typename A, typename E,
33 33         typename R, typename I, typename J>
34 34  template <typename K>
35 35  void
36 36  meldable_priority_queue<V, C, A, E, R, I, J>::insert(K b, K e) {
37 37      meldable_priority_queue q;
38 38      E* d;
39 39      try {
40 40          for (K c = b; c != e; ++c) {
41 41              d = create(*c);
42 42              I t(d);
43 43              (void) q.realizator.insert(t);
44 44          }
45 45      }
46 46      catch (...) {
47 47          destroy(d);
48 48          q.clear();
49 49          throw;
50 50      }
51 51      meld(q);
52 52  }
53 53
54 54  template <typename V, typename C, typename A, typename E,
55 55         typename R, typename I, typename J>
56 56  meldable_priority_queue<V, C, A, E, R, I, J>::meldable_priority_queue(

```

```

57     C const& comparator, A const& allocator)
58     : realizator(comparator), allocator(allocator) {
59 }
60
61 template <typename V, typename C, typename A, typename E,
62           typename R, typename I, typename J>
63 meldable_priority_queue<V, C, A, E, R, I, J>::meldable_priority_queue(
64     meldable_priority_queue const& other) {
65     meldable_priority_queue q;
66     try {
67         J first = other.begin();
68         J past_the_end = other.end();
69         q.insert(first, past_the_end);
70     }
71     catch (...) {
72         while (q.size() != 0) {
73             I t = q.realizator.extract();
74             destroy(t);
75         }
76         throw;
77     }
78     (*this).swap(q);
79 }
80
81 template <typename V, typename C, typename A, typename E,
82           typename R, typename I, typename J>
83 typename meldable_priority_queue<V, C, A, E, R, I, J>::container_type &
84 meldable_priority_queue<V, C, A, E, R, I, J>::operator=(
85     meldable_priority_queue const& other) {
86     meldable_priority_queue q;
87     try {
88         J first = other.begin();
89         J past_the_end = other.end();
90         q.insert(first, past_the_end);
91     }
92     catch (...) {
93         while (q.size() != 0) {
94             I t = q.realizator.extract();
95             destroy(t);
96         }
97         throw;
98     }
99     (*this).swap(q);
100    while (q.size() != 0) {
101        I t = q.realizator.extract();
102        destroy(t);
103    }
104    return *this;
105 }
106
107 template <typename V, typename C, typename A, typename E,
108           typename R, typename I, typename J>
109 meldable_priority_queue<V, C, A, E, R, I, J>::~meldable_priority_queue() {
110     clear();
111 }
112
113 template <typename V, typename C, typename A, typename E,
114           typename R, typename I, typename J>
115 A
116 meldable_priority_queue<V, C, A, E, R, I, J>::get_allocator() const {
117     return A(allocator);
118 }
119
120 template <typename V, typename C, typename A, typename E,
121           typename R, typename I, typename J>

```

```

122 C
123 meldable_priority_queue<V, C, A, E, R, I, J>::get_comparator() const {
124     return realizator.get_comparator();
125 }
126
127 template <typename V, typename C, typename A, typename E,
128           typename R, typename I, typename J>
129 I
130 meldable_priority_queue<V, C, A, E, R, I, J>::begin() {
131     return I(realizator.begin());
132 }
133
134 template <typename V, typename C, typename A, typename E,
135           typename R, typename I, typename J>
136 J
137 meldable_priority_queue<V, C, A, E, R, I, J>::begin() const {
138     return J(realizator.begin());
139 }
140
141 template <typename V, typename C, typename A, typename E,
142           typename R, typename I, typename J>
143 I
144 meldable_priority_queue<V, C, A, E, R, I, J>::end() {
145     return I(realizator.end());
146 }
147
148 template <typename V, typename C, typename A, typename E,
149           typename R, typename I, typename J>
150 J
151 meldable_priority_queue<V, C, A, E, R, I, J>::end() const {
152     return J(realizator.end());
153 }
154
155 template <typename V, typename C, typename A, typename E,
156           typename R, typename I, typename J>
157 bool
158 meldable_priority_queue<V, C, A, E, R, I, J>::empty() const {
159     return (*this).size() == size_type(0);
160 }
161
162 template <typename V, typename C, typename A, typename E,
163           typename R, typename I, typename J>
164 typename meldable_priority_queue<V, C, A, E, R, I, J>::size_type
165 meldable_priority_queue<V, C, A, E, R, I, J>::size() const {
166     return realizator.size();
167 }
168
169 template <typename V, typename C, typename A, typename E,
170           typename R, typename I, typename J>
171 typename meldable_priority_queue<V, C, A, E, R, I, J>::size_type
172 meldable_priority_queue<V, C, A, E, R, I, J>::max_size() const {
173     return realizator.max_size();
174 }
175
176 template <typename V, typename C, typename A, typename E,
177           typename R, typename I, typename J>
178 J
179 meldable_priority_queue<V, C, A, E, R, I, J>::top() const {
180     return J(realizator.top());
181 }
182
183 template <typename V, typename C, typename A, typename E,
184           typename R, typename I, typename J>
185 I
186 meldable_priority_queue<V, C, A, E, R, I, J>::top() {

```

```

187     return I(realizator.top());
188 }
189
190 template <typename V, typename C, typename A, typename E,
191          typename R, typename I, typename J>
192 I
193 meldable_priority_queue<V, C, A, E, R, I, J>::push(V const & v) {
194     E* p = create(v);
195     I t(p);
196     t = realizator.insert(t);
197     return t;
198 }
199
200 template <typename V, typename C, typename A, typename E,
201          typename R, typename I, typename J>
202 void
203 meldable_priority_queue<V, C, A, E, R, I, J>::pop() {
204     I t = realizator.top();
205     (void) realizator.extract(t);
206     destroy(t);
207 }
208
209 template <typename V, typename C, typename A, typename E,
210          typename R, typename I, typename J>
211 void
212 meldable_priority_queue<V, C, A, E, R, I, J>::erase(I t) {
213     (void) realizator.extract(t);
214     destroy(t);
215 }
216
217 template <typename V, typename C, typename A, typename E,
218          typename R, typename I, typename J>
219 void
220 meldable_priority_queue<V, C, A, E, R, I, J>::increase(I t, V const & v) {
221     realizator.increase(t, v);
222 }
223
224 template <typename V, typename C, typename A, typename E,
225          typename R, typename I, typename J>
226 void
227 meldable_priority_queue<V, C, A, E, R, I, J>::clear() {
228     while (realizator.size() != 0) {
229         I t = realizator.extract();
230         destroy(t);
231     }
232 }
233
234 template <typename V, typename C, typename A, typename E,
235          typename R, typename I, typename J>
236 void
237 meldable_priority_queue<V, C, A, E, R, I, J>::meld(meldable_priority_queue & q)
238 {
239     realizator.meld(q.realizator);
240 }
241
242 template <typename V, typename C, typename A, typename E,
243          typename R, typename I, typename J>
244 void
245 meldable_priority_queue<V, C, A, E, R, I, J>::swap(meldable_priority_queue & q)
246 {
247     realizator.swap(q.realizator);
248 }
249
250 template <typename V, typename C, typename A, typename E,
251          typename R, typename I, typename J>

```

```

250 meldable_priority_queue<V, C, A, E, R, I, J>&
251 meld(meldable_priority_queue<V, C, A, E, R, I, J>& r,
252      meldable_priority_queue<V, C, A, E, R, I, J>& s) {
253     r.meld(s.realizator);
254     return r;
255 }
256
257 template <typename V, typename C, typename A, typename E,
258          typename R, typename I, typename J>
259 void swap(meldable_priority_queue<V, C, A, E, R, I, J>& r,
260          meldable_priority_queue<V, C, A, E, R, I, J>& s) {
261     r.swap(s.realizator);
262 }
263 }

```

§ 3 *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 © 2009
13  */
14
15  #ifndef __CPHSTL_PRIRIOTY_QUEUE_ITERATOR__
16  #define __CPHSTL_PRIRIOTY_QUEUE_ITERATOR__
17
18  #include <cstdlib> // std::ptrdiff_t
19  #include <iterator> // std::forward_iterator_tag
20  #include <string> // std::string
21
22  namespace {
23      // if statement for compile-time meta-programming
24
25      template <bool, typename T, typename U>
26      class if_then_else;
27
28      template <typename T, typename U>
29      class if_then_else<true, T, U> {
30      public:
31          typedef T type;
32      };
33
34      template <typename T, typename U>
35      class if_then_else<false, T, U> {
36      public:
37          typedef U type;
38      };
39  }
40
41  namespace cphstl {
42      template <typename V, typename C, typename A, typename F,
43              typename R, typename I, typename J>
44      class meldable_priority_queue;
45
46      template <typename E, typename R, bool is_const = false>
47      class priority_queue_iterator {

```

```

48 public:
49
50     // types
51
52     typedef std::forward_iterator_tag iterator_category;
53     typedef E encapsulator_type;
54     typedef R realizator_type;
55     typedef typename E::element_type element_type;
56     typedef std::ptrdiff_t difference_type;
57
58     typedef typename if_then_else<is_const, element_type const*, element_type*>::
59         type pointer;
60     typedef typename if_then_else<is_const, element_type const &, element_type &
61         >::type reference;
62
63     typedef priority_queue_iterator<E, R, !is_const> complement;
64
65 private:
66
67     typedef typename if_then_else<is_const, E const*, E*>::type node_pointer;
68
69 public:
70
71     // friends
72
73     friend class priority_queue_iterator<E, R, !is_const>;
74
75     template <typename F, typename Q, bool both>
76     friend std::ostream & operator<<(std::ostream &, priority_queue_iterator<F, Q
77         , both> const &);
78
79     template <typename V, typename C, typename A, typename F, typename S,
80         typename I, typename J>
81     friend class cphstl::meldable_priority_queue;
82
83     // structors
84
85     priority_queue_iterator();
86
87     template <bool both>
88     priority_queue_iterator(priority_queue_iterator<E, R, both> const &);
89
90     template <bool both>
91     priority_queue_iterator & operator=(priority_queue_iterator<E, R, both> const
92         &);
93
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:

```

```

109
110     // converters to be used by the friends
111
112     priority_queue_iterator(node_pointer);
113     operator node_pointer() const;
114     operator std::string() const;
115
116     node_pointer link;
117 };
118 }
119
120 #include "priority-queue-iterator.i++"
121 #endif

```

§ 4 *priority-queue-iterator.i++*

```

1 /*
2  * Implementation of cphstl::priority_queue_iterator
3  *
4  * Author: Jyrki Katajainen © 2009, 2010
5  */
6
7 #include <sstream> // defines std::stringstream
8
9 namespace cphstl {
10     // default constructor
11
12     template <typename E, typename R, bool is_const>
13     priority_queue_iterator<E, R, is_const>::priority_queue_iterator()
14         : link(0) {
15     }
16
17     // copy constructor
18
19     template <typename E, typename R, bool is_const>
20     template <bool both>
21     priority_queue_iterator<E, R, is_const>::priority_queue_iterator(
22         priority_queue_iterator<E, R, both> const & a)
23         : link(a.link) {
24     }
25
26     // assignment
27
28     template <typename E, typename R, bool is_const>
29     template <bool both>
30     priority_queue_iterator<E, R, is_const>&
31     priority_queue_iterator<E, R, is_const>::operator=(priority_queue_iterator<E, R
32         , both> const & a) {
33         link = a.link;
34         return *this;
35     }
36
37     // destructor
38
39     template <typename E, typename R, bool is_const>
40     priority_queue_iterator<E, R, is_const>::~~priority_queue_iterator() {
41     }
42
43     // operator*
44
45     template <typename E, typename R, bool is_const>
46     typename priority_queue_iterator<E, R, is_const>::reference
47     priority_queue_iterator<E, R, is_const>::operator*() const {
48         return reference((*link).element());
49     }

```

```

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     node_pointer s = (*link).successor();
63     if (s != 0) {
64         link = s;
65         return *this;
66     }
67     node_pointer r = (*link).root();
68     typedef typename R::heap_store_type H;
69     typedef typename H::branch_type P;
70     P* p = (P*) (*r).owner();
71     P* q = (*p).successor();
72     if (q == 0) {
73         link = 0;
74         return *this;
75     }
76     link = (E*) (*q).child(1);
77     return *this;
78 }
79
80 // operator++; post-increment
81
82 template <typename E, typename R, bool is_const>
83 priority_queue_iterator<E, R, is_const>
84 priority_queue_iterator<E, R, is_const>::operator++(int) {
85     priority_queue_iterator<E, R, is_const> temporary(*this);
86     ++(*this);
87     return temporary;
88 }
89
90 // operator==
91
92 template <typename E, typename R, bool is_const>
93 template <bool both>
94 bool
95 priority_queue_iterator<E, R, is_const>::operator==(priority_queue_iterator<E,
96     R, both> const & a) const {
97     return link == a.link;
98 }
99 // operator!=
100
101 template <typename E, typename R, bool is_const>
102 template <bool both>
103 bool
104 priority_queue_iterator<E, R, is_const>::operator!=(priority_queue_iterator<E,
105     R, both> const & a) const {
106     return link != a.link;
107 }
108 // parametrized constructor
109

```

```

110 template <typename E, typename R, bool both>
111 priority_queue_iterator<E, R, both>::priority_queue_iterator(node_pointer p)
112     : link(p) {
113 }
114
115 // conversion operators
116
117 template <typename E, typename R, bool is_const>
118 priority_queue_iterator<E, R, is_const>::operator node_pointer() const {
119     return link;
120 }
121
122 template <typename E, typename R, bool is_const>
123 priority_queue_iterator<E, R, is_const>::operator std::string() const {
124     std::stringstream ss;
125     std::string address;
126     ss << (int)(char*)((*this).link);
127     ss >> address;
128
129     if (is_const == false) {
130         return std::string("iterator: node at ") + address;
131     }
132     else {
133         return std::string("const_iterator: node at ") + address;
134     }
135 }
136
137 // pipe to an output stream
138
139 template <typename E, typename R, bool both>
140 std::ostream&
141 operator<<(std::ostream& s, priority_queue_iterator<E, R, both> const& i) {
142     s << std::string(i);
143     return s;
144 }
145
146 }

```

§ 5 *comparator-proxy.h++*

```

1 /*
2  A proxy for a comparator
3
4  Authors: Jyrki Katajainen, Bo Simonsen © 2009, 2010
5 */
6
7 #ifndef __CPHSTL_COMPARATOR_PROXY__
8 #define __CPHSTL_COMPARATOR_PROXY__
9
10 namespace cphstl {
11     template <typename C>
12     class comparator_proxy {
13     public:
14
15         comparator_proxy(C const& c = C()) {
16             (*this).c = new C(c);
17         }
18
19         comparator_proxy(comparator_proxy const& cp) {
20             (*this).c = new C(*cp.c);
21         }
22
23         comparator_proxy operator=(comparator_proxy const& cp) {
24             delete (*this).c;

```

```

25     (*this).c = new C(*cp.c);
26     return (*this);
27 }
28
29     comparator_proxy operator=(C const & c) {
30         delete (*this).c;
31         (*this).c = new C(c);
32         return (*this);
33     }
34
35     ~comparator_proxy() {
36         delete (*this).c;
37     }
38
39     template <typename T, typename U>
40     bool operator()(T const & t, U const & u) const {
41         return (*c).operator()(t, u);
42     }
43
44     C subject() const {
45         return *c;
46     }
47
48     private:
49
50     C* c;
51 };
52 }
53
54 #endif

```

Transformations

§ 6 *relaxed-heap-modifier.h++*

```

1  /*
2   This modifier implements constant-time operations employed when
3   manipulating relaxed weak queues and relaxed weak heaps.
4
5   Authors: Stefan Edelkamp, Jyrki Katajainen, Jens Rasmussen ©
6   2009–2011
7  */
8
9  #ifndef __CPHSTL_RELAXED_HEAP_MODIFIER__
10 #define __CPHSTL_RELAXED_HEAP_MODIFIER__
11
12 #include <algorithm> // std::swap
13
14 #include "comparator-proxy.h++"
15
16 namespace cphstl {
17     template <typename N, typename C>
18     class relaxed_heap_modifier {
19     public:
20
21         typedef N node_type;
22         typedef C comparator_type;
23
24     protected:
25
26         comparator_proxy<C> comparator;
27
28     public:
29

```

```

30 relaxed_heap_modifier(C const& c = C())
31 : comparator(c) {
32 }
33
34 ~relaxed_heap_modifier() {
35 }
36
37 void swap(relaxed_heap_modifier& other) {
38     std::swap(comparator, other.comparator);
39 }
40
41 /*
42     p           q           p           p
43     / \        / \        / \        / \
44     -   -    -   -    -   -    -   -
45
46     */
47
48
49 template <typename L, typename V>
50 void add_leaf(N* p, N* q, L& level_registry, V& mark_registry) {
51     // level_registry.extract(p);
52     (*q).rank((*p).rank());
53     (*q).down();
54     (*q).parent(p);
55     if ((*p).right() == 0) {
56         (*p).right(q);
57     }
58     else {
59         (*p).left(q);
60     }
61     // level_registry.insert(p);
62     if ((*p).degree() == 2) {
63         level_registry.extract(p);
64     }
65
66     level_registry.insert(q);
67     mark_registry.insert(q, level_registry);
68 }
69
70 /*
71     p           p           p           q           p           q
72     / \        / \        / \        / \        / \        / \
73     -   -    -   -    -   -    -   -    -   -    -   -
74
75     */
76
77
78 template <typename L, typename V>
79 void cut_leaf(N* q, L& level_registry, V& mark_registry) {
80     level_registry.extract(q);
81     mark_registry.extract(q);
82     N* p = (*q).parent();
83     if (p != 0) {
84         level_registry.extract(p);
85         if ((*p).left() == q) {
86             (*p).left(0);
87         }
88         else if ((*p).right() == q) {
89             (*p).right(0);
90         }
91         else {
92         }
93         level_registry.insert(p);
94     }

```

```

95     (*q).make_singleton();
96 }
97
98 /*
99     a
100    |
101   p  q  →  a
102  / \  \    |
103 b   -  d  [q] p
104 */
105
106 void replace(N* p, N* q) {
107     // Assumption: p and q are not registered in any registries
108     N* a = (*p).parent();
109     N* b = (*p).left();
110     (*q).parent(a);
111     if (a != 0) {
112         if ((*a).left() == p) {
113             (*a).left(q);
114         }
115         else if ((*a).right() == p) {
116             (*a).right(q);
117         }
118         else {
119             }
120     }
121     (*q).left(b);
122     if (b != 0) {
123         (*b).parent(q);
124     }
125     (*p).make_singleton();
126 }
127
128 /*
129
130
131     p      r      →      p      r      r
132     / \    / \      / \    / \    / \
133    q  (s) q  (s)  q  (s)  q  (s)  q  (s)
134   / \  / \  / \  / \  / \  / \  / \
135  a  b c  d  a  b c  d  c  d a  b  c  d  a  b  c  d
136 */
137
138 template <typename L>
139 N* join(N* p, N* r, L& level_registry) {
140     // Assumption: p and q are not registered, but their decendants are
141     N* q = (*p).right();
142     N* s = (*r).right();
143     if (comparator((*p).element(), (*r).element())) {
144         if (s != 0 && (*s).is_marked()) {
145             N* a = 0;
146             if (q != 0) {
147                 a = (*q).left();
148                 level_registry.extract(q);
149             }
150             N* c = (*s).left();
151             level_registry.extract(s);
152             (*r).right(p);
153             (*p).parent(r);
154             (*p).left(q);
155             if (q != 0) {
156                 (*q).parent(p);
157                 (*q).left(c);
158                 if (c != 0) {
159                     (*c).parent(q);

```

```

160     }
161     }
162     (*p).right(s);
163     if (s != 0) {
164         (*s).parent(p);
165         (*s).left(a);
166         if (a != 0) {
167             (*a).parent(s);
168         }
169     }
170     (*r).up();
171     level_registry.insert(p);
172     if (q != 0) {
173         level_registry.insert(q);
174     }
175     level_registry.insert(s);
176     return r;
177 }
178 (*r).right(p);
179 (*p).parent(r);
180 (*p).left(s);
181 if (s != 0) {
182     (*s).parent(p);
183 }
184 (*r).up();
185 level_registry.insert(p);
186 return r;
187 }
188 (*p).right(r);
189 (*r).parent(p);
190 (*r).left(q);
191 if (q != 0) {
192     (*q).parent(r);
193 }
194 (*p).up();
195 level_registry.insert(r);
196 return p;
197 }
198
199 /*
200     a           a
201     |           |
202     p           q
203    / \         / \
204   b  c       b  c
205    \         /
206     ...     ...
207     /         /
208    e         e
209    / \       / \
210   q  g     p  g
211  / \     / \
212 f  g   f  g
213 */
214
215 template <typename L>
216 void promote(N* p, N* q, L& level_registry) {
217     level_registry.extract(p);
218     level_registry.extract(q);
219     N* a = (*p).parent();
220     N* b = (*p).left();
221     bool at_root = (*p).is_root();
222     N* c = (*p).right();
223     N* e = (*q).parent();
224     N* f = (*q).left();
225     N* g = (*q).right();

```

```

225 typename N::rank_type tmp = (*p).rank();
226 (*p).rank((*q).rank());
227 (*q).rank(tmp);
228 if (p == (*q).parent()) {
229     if (! at_root) {
230         if ((*a).left() == p) {
231             (*a).left(q);
232         }
233         else {
234             (*a).right(q);
235         }
236     }
237     (*q).parent(a);
238     (*q).left(b);
239     if (b != 0) {
240         (*b).parent(q);
241     }
242     (*p).parent(q);
243     (*p).left(f);
244     (*p).right(g);
245     (*q).right(p);
246     if (f != 0) {
247         (*f).parent(p);
248     }
249     if (g != 0) {
250         (*g).parent(p);
251     }
252 }
253 else {
254     if (! at_root) {
255         if ((*a).left() == p) {
256             (*a).left(q);
257         }
258         else {
259             (*a).right(q);
260         }
261     }
262     (*q).parent(a);
263     (*q).left(b);
264     if (b != 0) {
265         (*b).parent(q);
266     }
267     (*q).right(c);
268     (*c).parent(q);
269     (*p).parent(e);
270     (*e).left(p);
271     (*p).left(f);
272     (*p).right(g);
273     if (f != 0) {
274         (*f).parent(p);
275     }
276     if (g != 0) {
277         (*g).parent(p);
278     }
279 }
280 level_registry.insert(p);
281 level_registry.insert(q);
282 }
283
284 /*
285
286
287
288
289

```

$$\begin{array}{ccc}
 & p & \\
 & / \quad \backslash & \\
 [q] & & r \\
 / \quad \backslash & & / \quad \backslash \\
 a & b \quad c & d
 \end{array}
 \rightarrow
 \begin{array}{ccc}
 & p & \\
 & / \quad \backslash & \\
 r & & [q] \\
 / \quad \backslash & & / \quad \backslash \\
 a & d \quad c & b
 \end{array}$$

```

290  */
291
292  template <typename L, typename V>
293  void cleaning_transformation(N* q, L& level_registry, V& mark_registry) {
294      N* p = (*q).parent();
295      mark_registry.extract(q);
296      N* r = (*p).right();
297      if (r == 0) {
298          (*p).left(0);
299          (*p).right(q);
300          mark_registry.insert(q, level_registry);
301          return;
302      }
303      N* a = (*q).left();
304      N* c = (*r).left();
305      bool a_was_marked = a != 0 && (*a).is_marked();
306      bool c_was_marked = c != 0 && (*c).is_marked();
307      if ((a == 0) != (c == 0)) {
308          level_registry.extract(q);
309          level_registry.extract(r);
310      }
311      if (a_was_marked) {
312          mark_registry.extract(a);
313      }
314      if (c_was_marked) {
315          mark_registry.extract(c);
316      }
317      (*p).left(r);
318      (*p).right(q);
319      (*r).left(a);
320      (*q).left(c);
321      if (a != 0) {
322          (*a).parent(r);
323      }
324      if (c != 0) {
325          (*c).parent(q);
326      }
327      if ((a == 0) != (c == 0)) {
328          level_registry.insert(q);
329          level_registry.insert(r);
330      }
331      mark_registry.insert(q, level_registry);
332      if (a_was_marked) {
333          mark_registry.insert(a, level_registry);
334      }
335      if (c_was_marked) {
336          mark_registry.insert(c, level_registry);
337      }
338  }
339
340  /*
341
342      g
343      |
344      (p)
345     / \
346    a  [q]
347       / \
348      b  c
349
350      →
351
352      g
353      |
354      [q]
355     / \
356    a  p
357       / \
358      c  b
359
360      or
361
362      g
363      |
364      (p)
365     / \
366    a  q
367       / \
368      b  c
369
370  */
371
372  template <typename L, typename V>
373  void parent_transformation(N* q, L& level_registry, V& mark_registry) {
374      N* p = (*q).parent();
375      bool p_was_marked = p != 0 && (*p).is_marked();
376      N* g = (*p).parent();

```

```

355     N* a = (*p).left();
356     N* b = (*q).left();
357     N* c = (*q).right();
358     bool b_was_marked = b != 0 && (*b).is_marked();
359     bool c_was_marked = c != 0 && (*c).is_marked();
360     if (p_was_marked) {
361         mark_registry.extract(p);
362     }
363     mark_registry.extract(q);
364     if (b_was_marked) {
365         mark_registry.extract(b);
366     }
367     if (c_was_marked) {
368         mark_registry.extract(c);
369     }
370     if (comparator((*p).element(), (*q).element())) {
371         level_registry.extract(p);
372         level_registry.extract(q);
373         (*p).down();
374         (*q).up();
375         if (!(*p).is_root()) {
376             if ((*g).right() == p) {
377                 (*g).right(q);
378             }
379             else {
380                 (*g).left(q);
381             }
382         }
383         (*q).left(a);
384         if (a != 0) {
385             (*a).parent(q);
386         }
387         (*q).parent(g);
388         (*q).right(p);
389         (*p).parent(q);
390         (*p).left(c);
391         if (c != 0) {
392             (*c).parent(p);
393         }
394         (*p).right(b);
395         if (b != 0) {
396             (*b).parent(p);
397         }
398         level_registry.insert(p);
399         level_registry.insert(q);
400         if (!(*q).is_root()) {
401             mark_registry.insert(q, level_registry);
402         }
403     }
404     else {
405         if (p_was_marked) {
406             mark_registry.insert(p, level_registry);
407         }
408     }
409     if (b_was_marked) {
410         mark_registry.insert(b, level_registry);
411     }
412     if (c_was_marked) {
413         mark_registry.insert(c, level_registry);
414     }
415 }
416
417 /*
418     g           g           g
419     |           |           |

```

```

420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484

```

```

/*
template <typename L, typename V>
void sibling_transformation(N* q, L& level_registry, V& mark_registry) {
    N* p = (*q).parent();
    N* r = (*p).right();
    lift(q, level_registry, mark_registry);

    parent_transformation(r, level_registry, mark_registry);
}

/*

*/
template <typename L, typename V>
void zig_zag_transformation(N* r, L& level_registry, V& mark_registry) {
    N* q = (*r).parent();
    N* p = (*q).parent();
    mark_registry.extract(r);
    if (comparator((*p).element(), (*r).element())) {
        level_registry.extract(p);
        level_registry.extract(r);
        N* a = (*p).left();
        N* g = (*p).parent();
        N* b = (*r).left();
        N* c = (*r).right();
        bool b_was_marked = (b != 0) && (*b).is_marked();
        bool c_was_marked = (c != 0) && (*c).is_marked();
        (*p).down();
        (*p).down();
        (*r).up();
        (*r).up();
        if (!(*p).is_root()) {
            if ((*g).right() == p) {
                (*g).right(r);
            }
            else {
                (*g).left(r);
            }
        }
        (*r).left(a);
        if (a != 0) {
            (*a).parent(r);
        }
        (*r).parent(g);
        (*r).right(q);
        (*p).parent(q);
        (*q).parent(r);
        (*q).left(p);
        (*p).left(c);
        (*p).right(b);
    }
}

```

```

485     if (b != 0) {
486         (*b).parent(p);
487     }
488     if (c != 0) {
489         (*c).parent(p);
490     }
491     level_registry.insert(r);
492     level_registry.insert(p);
493     mark_registry.insert(r, level_registry);
494     if (b_was_marked) {
495         mark_registry.insert(b, level_registry);
496     }
497     if (c_was_marked) {
498         mark_registry.insert(c, level_registry);
499     }
500 }
501 }
502
503 /*
504     p           r           p           r           [q]           [s]
505     \          /          \          /          \          /          \
506     [q]       [s]       r       p       [q]       [s]
507     / \      / \      / \      / \      / \      / \
508     a  b    c  d    a  c    c  a    b  d    d  b
509 */
510
511 template <typename L, typename V>
512 void pair_transformation(N* q, N* s, L& level_registry, V& mark_registry) {
513     N* p = (*q).parent();
514     N* r = (*s).parent();
515     bool p_less_than_r = comparator((*p).element(), (*r).element());
516     switch (p_less_than_r) {
517     case 0 /* p >= r */:
518         cross(p, q, r, s, level_registry, mark_registry);
519         return;
520     case 1 /* p < r */:
521         cross(r, s, p, q, level_registry, mark_registry);
522         return;
523     default:
524     }
525 }
526
527 protected:
528
529 /*
530     g           g
531     |           |
532     p           [q]
533     / \        / \
534     [q] [r]   p   [r]
535     / \ / \   / \ / \
536     a b c d   a c b d
537 */
538
539
540 template <typename L, typename V>
541 void lift(N* q, L& level_registry, V& mark_registry) {
542     N* p = (*q).parent();
543     N* r = (*p).right();
544     N* a = (*q).left();
545     N* b = (*q).right();
546     N* c = (*r).left();
547     N* g = (*p).parent();
548     level_registry.extract(p);
549     level_registry.extract(q);

```

```

550     level_registry.extract(r);
551     mark_registry.extract(q);
552     mark_registry.extract(r);
553     bool a_was_marked = a != 0 && (*a).is_marked();
554     bool b_was_marked = b != 0 && (*b).is_marked();
555     bool c_was_marked = c != 0 && (*c).is_marked();
556     if (a_was_marked) {
557         mark_registry.extract(a);
558     }
559     if (b_was_marked) {
560         mark_registry.extract(b);
561     }
562     if (c_was_marked) {
563         mark_registry.extract(c);
564     }
565     (*p).down();
566     (*q).up();
567     (*q).parent(g);
568     if ((*g).right() == p) {
569         (*g).right(q);
570     }
571     else {
572         (*g).left(q);
573     }
574     (*q).left(p);
575     (*q).right(r);
576     (*p).parent(q);
577     (*r).parent(q);
578     (*p).left(a);
579     (*p).right(c);
580     (*r).left(b);
581     if (a != 0) {
582         (*a).parent(p);
583     }
584     if (b != 0) {
585         (*b).parent(r);
586     }
587     if (c != 0) {
588         (*c).parent(p);
589     }
590     level_registry.insert(p);
591     level_registry.insert(q);
592     level_registry.insert(r);
593     mark_registry.insert(q, level_registry);
594     mark_registry.insert(r, level_registry);
595     if (a_was_marked) {
596         mark_registry.insert(a, level_registry);
597     }
598     if (b_was_marked) {
599         mark_registry.insert(b, level_registry);
600     }
601     if (c_was_marked) {
602         mark_registry.insert(c, level_registry);
603     }
604 }
605
606 /*
607     p           r           p           [q]           [s]
608     \          /          \          / \          / \          / \
609     [q]       [s]       r       [q] [s]       x       q       x       q
610     / \      / \      / \      / \      / \      / \      / \
611     a  b    c  d    a  c    b  d    b  d    d  b
612 */
613
614 template <typename L, typename V>

```

```

615 void cross(N* p, N* q, N* r, N* s, L& level_registry, V& mark_registry) {
616     // Precondition: p  $\succcurlyeq$  r
617     N* a = (*q).left();
618     N* b = (*q).right();
619     N* c = (*s).left();
620     N* d = (*s).right();
621     N* g = (*r).parent();
622     N* x = (*r).left();
623     bool a_was_marked = (a != 0) && (*a).is_marked();
624     bool b_was_marked = (b != 0) && (*b).is_marked();
625     bool c_was_marked = (c != 0) && (*c).is_marked();
626     bool d_was_marked = (d != 0) && (*d).is_marked();
627     level_registry.extract(q);
628     level_registry.extract(r);
629     level_registry.extract(s);
630     mark_registry.extract(q);
631     mark_registry.extract(s);
632     if (a_was_marked) {
633         mark_registry.extract(a);
634     }
635     if (b_was_marked) {
636         mark_registry.extract(b);
637     }
638     if (c_was_marked) {
639         mark_registry.extract(c);
640     }
641     if (d_was_marked) {
642         mark_registry.extract(d);
643     }
644     (*r).down();
645     (*r).parent(p);
646     (*r).left(a);
647     (*r).right(c);
648     (*p).right(r);
649     if (a != 0) {
650         (*a).parent(r);
651     }
652     if (c != 0) {
653         (*c).parent(r);
654     }
655
656     bool q_less_than_s = comparator((*q).element(), (*s).element());
657     if (q_less_than_s) {
658         (*s).up();
659         if (g != 0) {
660             if ((*g).right() == r) {
661                 (*g).right(s);
662             }
663             else {
664                 (*g).left(s);
665             }
666         }
667         (*s).parent(g);
668         (*s).left(x);
669         if (x != 0) {
670             (*x).parent(s);
671         }
672         (*s).right(q);
673         (*q).parent(s);
674         (*q).left(d);
675         if (d != 0) {
676             (*d).parent(q);
677         }
678         if (!(*s).is_root()) {
679             mark_registry.insert(s, level_registry);

```

```

680     }
681   }
682   else {
683     (*q).up();
684     if (g != 0) {
685       if ((*g).right() == r) {
686         (*g).right(q);
687       }
688       else {
689         (*g).left(q);
690       }
691     }
692     (*q).parent(g);
693     (*q).left(x);
694     if (x != 0) {
695       (*x).parent(q);
696     }
697     (*q).right(s);
698     (*s).parent(q);
699     (*s).left(b);
700     if (b != 0) {
701       (*b).parent(s);
702     }
703     if (! (*q).is_root()) {
704       mark_registry.insert(q, level_registry);
705     }
706   }
707   level_registry.insert(q);
708   level_registry.insert(r);
709   level_registry.insert(s);
710   if (a_was_marked) {
711     mark_registry.insert(a, level_registry);
712   }
713   if (b_was_marked) {
714     mark_registry.insert(b, level_registry);
715   }
716   if (c_was_marked) {
717     mark_registry.insert(c, level_registry);
718   }
719   if (d_was_marked) {
720     mark_registry.insert(d, level_registry);
721   }
722 }
723 };
724 }
725
726 #endif

```

§ 7 naive-mark-registry.h++

```

1  /*
2  2  A naive mark registry; a marked node must be removed before any new
3  3  marked nodes are introduced.
4  4
5  5  Author: Jyrki Katajainen © 2009–2011
6  6  */
7  7
8  8  #ifndef __CPHSTL_NAIVE_MARK_REGISTRY__
9  9  #define __CPHSTL_NAIVE_MARK_REGISTRY__
10 10
11 11 #include <algorithm> // std::swap
12 12
13 13 #include "comparator-proxy.h++"
14 14 #include <cstdint> // std::size_t

```

```

15
16 namespace cphstl {
17     template <typename N, typename M, typename C>
18     class naive_mark_registry {
19     public:
20
21         typedef N node_type;
22         typedef M modifier_type;
23         typedef C comparator_type;
24         typedef std::size_t size_type;
25
26     protected:
27
28         comparator_proxy<C> comparator;
29         M modifier;
30
31     public:
32
33         explicit naive_mark_registry(C const& c = C())
34             : comparator(c), modifier(c) {
35         }
36
37         ~naive_mark_registry() {
38         }
39
40         bool empty() const {
41             return true;
42         }
43
44         N* __first__() const {
45             return (N*) 0;
46         }
47
48         int size() {
49             return 0;
50         }
51
52         N* top() const {
53             return (N*) 0;
54         }
55
56         template <typename L>
57         void clear(L& level_registry) {
58             return;
59         }
60
61         template <typename L>
62         void insert(N* q, L& level_registry) {
63             if ((*q).is_root()) {
64                 return;
65             }
66             immediate_fix(q, level_registry);
67         }
68
69         void extract(N* p) {
70         }
71
72         template <typename L>
73         void reduce(L&) {
74         }
75
76         void swap(naive_mark_registry& other) {
77             std::swap(comparator, other.comparator);
78         }
79

```

```

80 protected:
81
82     template <typename L>
83     void immediate_fix (N* q, L& level_registry) {
84         N* p = (*q).distinguished_ancestor();
85         while (p != 0) {
86             if (comparator((*p).element(), (*q).element())) {
87                 modifier.promote(p, q, level_registry);
88                 p = (*q).distinguished_ancestor();
89             }
90             else {
91                 return;
92             }
93         }
94     }
95 };
96 }
97
98 #endif

```

§ 8 eager-mark-registry.h++

```

1  /*
2  2  An eager mark registry keeps track of the marked nodes in a
3  3  rank-relaxed heap. This registry is a duplicate registry with the
4  4  following characteristics:
5  5
6  6  - key: rank stored at the nodes
7  7  - duplicate linking: none since there are no duplicates, except
8  8  perhaps at one level
9  9  - access: a marked node of some specific rank
10 10 - duplicate reduction: maintain the following invariants:
11 11
12 12 1) A marked node does not have any neighbours (parent, child, or
13 13 sibling) that are marked, neither there exists a marked node of
14 14 the same rank.
15 15
16 16 2) A marked node is always a right child of its parent.
17 17
18 18 Authors: Stefan Edelkamp, Jyrki Katajainen © 2009–2011
19 19 */
20
21 #ifndef __CPHSTL_EAGER_MARK_REGISTRY__
22 #define __CPHSTL_EAGER_MARK_REGISTRY__
23
24 #include <algorithm> // std::swap
25
26 #include "comparator-proxy.h++"
27 #include <climits> // CHAR_BIT
28 #include <vector>
29
30 namespace cphstl {
31     template <typename N, typename M, typename C>
32     class eager_mark_registry {
33     public:
34
35         typedef N node_type;
36         typedef M modifier_type;
37         typedef C comparator_type;
38         typedef unsigned char rank_type;
39
40     protected:
41
42         typedef unsigned long long W;

```

```

43     enum {word_size = CHAR_BIT * sizeof(W)};
44
45     comparator_proxy<C> comparator;
46     M modifier;
47     W occupied;
48     bool invariants_ok;
49     std::vector<N*> header;
50
51 public:
52
53     eager_mark_registry(C const & c = C())
54         : comparator(c), modifier(c), occupied(0), invariants_ok(true), header() {
55         header.resize(word_size, (N*) 0);
56     }
57
58     ~eager_mark_registry() {
59     }
60
61     bool empty() const {
62         return occupied == 0;
63     }
64
65     N* __first__() const {
66         if (occupied == 0) {
67             return (N*) 0;
68         }
69         rank_type h = most_significant_one(occupied);
70         return header[h];
71     }
72
73     rank_type size() const {
74         return __builtin_popcountl(occupied);
75     }
76
77     template<typename L>
78     void clear(L & level_registry) {
79         while (size() > 0) {
80             rank_type rank = least_significant_one(occupied);
81             N* q = header[rank];
82             N* p = (*q).parent();
83             extract(q);
84             modifier.parent_transformation(q, level_registry, *this);
85             if ((*p).right() == q) {
86                 q = p;
87             }
88             insert(q, level_registry);
89         }
90     }
91
92     N* top() const {
93         N* maximum = 0;
94         W copy = occupied;
95         while (copy != 0) {
96             rank_type h = most_significant_one(copy);
97             copy = unset(copy, h);
98             N* q = header[h];
99
100             if (maximum == 0)
101                 maximum = q;
102             else {
103                 if (comparator((*maximum).element(), (*q).element())) {
104                     maximum = q;
105                 }
106             }
107         }

```

```

108     return maximum;
109 }
110
111 template <typename L>
112 void insert(N* q, L& level_registry) {
113     if ((*q).is_marked()) {
114         return;
115     }
116     (*q).arena_index(0); // marking is denoted like this
117     if (! invariants_ok) {
118         return; // no-op
119     }
120     invariants_ok = false;
121     N* r = (*q).right();
122     if (r != 0 && (*r).is_marked()) {
123         modifier.parent_transformation(r, level_registry, *this);
124         if ((*q).right() != r) {
125             q = r;
126         }
127     }
128     while (true) {
129         if ((*q).is_root()) {
130             (*q).arena_index(0xff);
131             invariants_ok = true;
132             return;
133         }
134         N* p = (*q).parent();
135         if ((*p).is_root()) {
136             modifier.parent_transformation(q, level_registry, *this);
137             invariants_ok = true;
138             return;
139         }
140         if ((*p).left() == q) {
141             N* s = (*p).right();
142             if (s != 0 && (*s).is_marked()) {
143                 modifier.sibling_transformation(q, level_registry, *this);
144                 if ((*s).right() == q) {
145                     q = s;
146                     continue;
147                 }
148                 else if ((*q).right() == s) {
149                     continue;
150                 }
151                 else {
152                     }
153             }
154             if ((*p).is_marked()) {
155                 modifier.zig_zag_transformation(q, level_registry, *this);
156                 if ((*p).left() == q) {
157                     invariants_ok = true;
158                     return;
159                 }
160             }
161             modifier.parent_transformation(p, level_registry, *this);
162             if ((*p).right() == q) {
163                 q = p;
164             }
165             continue;
166         }
167         modifier.cleaning_transformation(q, level_registry, *this);
168     }
169     if ((*p).is_marked()) {
170         modifier.parent_transformation(q, level_registry, *this);
171         if ((*p).right() == q) {
172             q = p;

```

```

173     }
174     break;
175 }
176 rank_type h = (*q).rank();
177 N* s = header[h];
178 if (s != 0) {
179     modifier.pair_transformation(q, s, level_registry, *this);
180     if ((*s).right() == q) {
181         q = s;
182     }
183     continue;
184 }
185 break;
186 }
187 rank_type h = (*q).rank();
188 occupied = set(occupied, h);
189 header[h] = q;
190 invariants_ok = true;
191 }
192
193 void extract(N* q) {
194     if (! (*q).is_marked()) {
195         return;
196     }
197     rank_type h = (*q).rank();
198     if (header[h] == q) {
199         occupied = unset(occupied, h);
200         header[h] = 0;
201     }
202     else {
203     }
204     (*q).arena_index(0xff);
205 }
206
207 template <typename L>
208 void reduce(L&) {
209 }
210
211 void swap(eager_mark_registry& other) {
212     std::swap(comparator, other.comparator);
213     modifier.swap(other.modifier);
214     std::swap(occupied, other.occupied);
215     header.swap(other.header);
216 }
217
218 protected:
219
220 bool get(W word, rank_type i) const {
221     word = word >> i;
222     return word & 1ULL;
223 }
224
225 rank_type least_significant_one(W word) const {
226     return __builtin_ctz1(word); // GNU gcc specific
227 }
228
229 rank_type most_significant_one(W word) const {
230     return word_size - __builtin_clz1(word) - 1; // GNU gcc specific
231 }
232
233 W set(W word, rank_type i) const {
234     // Assumption: the bit at position i is 0
235     word += (1ULL << i);
236     return word;
237 }

```

```

238
239     W unset(W word, rank_type i) const {
240         // Assumption: the bit at position i is 1
241         word -= (1ULL << i);
242         return word;
243     }
244 };
245 }
246
247 #endif

```

§ 9 lazy-mark-registry.h++

```

1  /*
2   A lazy mark registry keeps track of the marked nodes in a
3   run-relaxed heap. This registry is a duplicate registry with the
4   following characteristics:
5
6   - key: rank stored or deduced from the information stored at the nodes
7   - duplicate linking: maintain references to the marked nodes in a
8     small contiguous segment of memory called an arena; the nodes
9     themselves store an index to this arena, each occupying  $O(\lg n)$ 
10    bits (which can be held in a byte)
11  - access: a marked node of the given rank; marked nodes of type I and II
12  - duplicate reduction: divide the marked nodes into two categories:
13
14    I: A marked node belongs to this category if its parent is also
15    marked. If such a marked node exists, the number of marked nodes
16    can be reduced by applying the zig-zag and parent transformations,
17    or the parent, cleaning, and sibling transformations. However, a
18    marked node of the smallest rank of this type must be selected to
19    make the reduction possible (grandparent must be non-marked).
20
21    II: A marked node belongs to this category if its parent is
22    non-marked. If two such marked nodes exist, the number of marked
23    nodes can be reduced by applying the cleaning, pair, and parent
24    transformations.
25
26    Author: Jyrki Katajainen © 2009–2011
27  */
28
29 #ifndef __CPHSTL_LAZY_MARK_REGISTRY__
30 #define __CPHSTL_LAZY_MARK_REGISTRY__
31
32 #include <algorithm> // std::swap
33
34 #include "comparator-proxy.h++"
35 #include <climits> // CHAR_BIT
36 #include <cstddef> // std::size_t
37 #include <vector>
38
39 namespace cphstl {
40     template <typename N, typename M, typename C>
41     class lazy_mark_registry {
42     public:
43
44         typedef N node_type;
45         typedef M modifier_type;
46         typedef C comparator_type;
47         typedef unsigned char rank_type;
48         typedef unsigned char index_type;
49         typedef std::size_t size_type;
50
51     protected:

```

```

52
53 typedef unsigned int bitvector_type;
54 enum {word_size = CHAR_BIT * sizeof(bitvector_type)};
55
56 struct entry_type {
57     N* marked_node;
58     index_type next;
59     index_type previous;
60 };
61
62 typedef std::vector<entry_type> arena_type;
63 typedef std::vector<N*> header_type;
64
65 comparator_proxy<C> comparator;
66 M modifier;
67 bitvector_type occupied;
68 bitvector_type category_I;
69 bitvector_type more_than_one;
70 bitvector_type free;
71 header_type header;
72 arena_type arena;
73
74 public:
75
76 lazy_mark_registry(C const& c = C())
77     : comparator(c), modifier(c), occupied(0), category_I(0),
78       more_than_one(0), free(~ bitvector_type(0)), header(), arena() {
79     header.resize(word_size, (N*) 0);
80     arena.resize(word_size);
81 }
82
83 ~lazy_mark_registry() {
84 }
85
86 bool empty() const {
87     return occupied == 0;
88 }
89
90 N* __first__() const {
91     if (occupied == 0) {
92         return (N*) 0;
93     }
94     rank_type h = most_significant_one(occupied);
95     return header[h];
96 }
97
98 N* top() const {
99     bitvector_type used = ~free;
100     if (used == 0) {
101         return (N*) 0;
102     }
103     N* maximum = 0;
104     while (used != 0) {
105         index_type i = most_significant_one(used);
106         N* q = arena[i].marked_node;
107         if (maximum == 0) {
108             maximum = q;
109         }
110         else if (q != 0 && comparator((*maximum).element(), (*q).element())) {
111             maximum = q;
112         }
113         used = unset(used, i);
114     }
115     return maximum;
116 }

```

```

117
118     rank_type size() const {
119         return __builtin_popcountl(occupied);
120     }
121
122     template <typename L>
123     void clear(L& level_registry) {
124         return; // dummy
125     }
126
127     template <typename L>
128     void insert(N* p, L&) {
129         if ((*p).is_marked() || (*p).is_root()) {
130             return;
131         }
132         rank_type h = (*p).rank();
133         occupied = set(occupied, h);
134         index_type i = allocate(arena, free);
135         create(arena, i, p);
136         N* a = (*p).parent();
137         if (a != 0 && (*a).is_marked()) {
138             push_front(header, p, arena);
139             category_I = set(category_I, h);
140         }
141         else {
142             push_back(header, p, arena);
143         }
144         if (front(header, h, arena) != back(header, h, arena)) {
145             more_than_one = set(more_than_one, h);
146         }
147         N* b = (*p).left();
148         if ((b != 0) && (*b).is_marked()) {
149             erase(header, b, arena);
150             push_front(header, b, arena);
151             category_I = set(category_I, (*b).rank());
152         }
153         N* c = (*p).right();
154         if ((c != 0) && (*c).is_marked()) {
155             erase(header, c, arena);
156             push_front(header, c, arena);
157             category_I = set(category_I, (*c).rank());
158         }
159     }
160
161     void extract(N* q) {
162         if (!(*q).is_marked()) {
163             return;
164         }
165         index_type j = (*q).arena_index();
166         erase(header, q, arena);
167         destroy(arena, q);
168         deallocate(arena, j, free);
169         rank_type h = (*q).rank();
170         N* p = front(header, h, arena);
171         if (p == 0) {
172             occupied = unset(occupied, h);
173         }
174         if (p == 0 || back(header, h, arena) == p) {
175             more_than_one = unset(more_than_one, h);
176         }
177         if (p == 0) {
178             category_I = unset(category_I, h);
179         }
180         else {
181             N* u = (*p).parent();

```

```

182         if (!(*u).is_marked()) {
183             category_I = unset(category_I, h);
184         }
185     }
186     N* b = (*q).left();
187     bool change = false;
188     if (b != 0 && (*b).is_marked()) {
189         change = true;
190         erase(header, b, arena);
191         push_back(header, b, arena);
192     }
193     N* c = (*q).right();
194     if (c != 0 && (*c).is_marked()) {
195         change = true;
196         erase(header, c, arena);
197         push_back(header, c, arena);
198     }
199     if (change) {
200         rank_type g = (b != 0) ? (*b).rank() : (*c).rank();
201         N* r = front(header, g, arena);
202         if (!((*r).parent() → is_marked())) {
203             category_I = unset(category_I, g);
204         }
205     }
206 }
207
208 template <typename L>
209 void reduce(L& level_registry) {
210     if (category_I != 0) {
211         rank_type h = most_significant_one(category_I);
212         N* r = front(header, h, arena);
213         N* q = (*r).parent();
214         if ((*q).right() == r) {
215             modifier.parent_transformation(r, level_registry, *this);
216             return;
217         }
218         N* p = (*q).parent();
219         if ((*p).right() == q) {
220             modifier.zig_zag_transformation(r, level_registry, *this);
221             if ((*q).left() == p) {
222                 modifier.parent_transformation(q, level_registry, *this);
223             }
224             return;
225         }
226         N* s = (*p).right();
227         if ((*s).is_marked()) {
228             modifier.sibling_transformation(q, level_registry, *this);
229             return;
230         }
231         modifier.cleaning_transformation(q, level_registry, *this);
232         N* x = (*s).right();
233         if (x != 0 && (*x).is_marked()) {
234             modifier.sibling_transformation(r, level_registry, *this);
235             return;
236         }
237         modifier.cleaning_transformation(r, level_registry, *this);
238         modifier.parent_transformation(r, level_registry, *this);
239         if ((*p).left() == r) {
240             modifier.sibling_transformation(r, level_registry, *this);
241         }
242         return;
243     }
244     if (more_than_one == 0) {
245         return;
246     }

```

```

247     rank_type h = most_significant_one(more_than_one);
248     N* q = front(header, h, arena);
249     N* s = back(header, h, arena);
250     N* p = (*q).parent();
251     N* r = (*s).parent();
252     if ((*p).is_marked()) {
253         if ((*p).left() == q) {
254             modifier.zig_zag_transformation(q, level_registry, *this);
255             if ((*p).left() != q) {
256                 modifier.parent_transformation(p, level_registry, *this);
257             }
258             return;
259         }
260         modifier.parent_transformation(q, level_registry, *this);
261         return;
262     }
263     if ((*r).is_marked()) {
264         if ((*r).left() == s) {
265             modifier.zig_zag_transformation(s, level_registry, *this);
266             if ((*r).left() != s) {
267                 modifier.parent_transformation(r, level_registry, *this);
268             }
269             return;
270         }
271         modifier.parent_transformation(s, level_registry, *this);
272         return;
273     }
274     if ((*p).left() == q) {
275         N* x = (*p).right();
276         if (x != 0 && (*x).is_marked()) {
277             modifier.sibling_transformation(q, level_registry, *this);
278             return;
279         }
280         modifier.cleaning_transformation(q, level_registry, *this);
281     }
282     if ((*r).left() == s) {
283         N* y = (*r).right();
284         if (y != 0 && (*y).is_marked()) {
285             modifier.sibling_transformation(s, level_registry, *this);
286             return;
287         }
288         modifier.cleaning_transformation(s, level_registry, *this);
289     }
290     modifier.pair_transformation(q, s, level_registry, *this);
291 }
292
293 void swap(lazy_mark_registry& other) {
294     std::swap(comparator, other.comparator);
295     modifier.swap(other.modifier);
296     std::swap(occupied, other.occupied);
297     std::swap(category_I, other.category_I);
298     std::swap(more_than_one, other.more_than_one);
299     header.swap(other.header);
300     arena.swap(other.arena);
301 }
302
303 protected:
304
305     index_type allocate(arena_type& arena, bitvector_type& free) {
306         index_type i = most_significant_one(free);
307         free = unset(free, i);
308         return i;
309     }
310
311     void create(arena_type& arena, index_type i, N* p) {

```

```

312     (*p).arena_index(i);
313     arena[i].marked_node = (N*) 0;
314     arena[i].previous = 0xff;
315     arena[i].next = 0xff;
316 }
317
318 void deallocate(arena_type & arena, index_type i, bitvector_type & free) {
319     free = set(free, i);
320 }
321
322 void destroy(arena_type & arena, N* p) {
323     index_type i = (*p).arena_index();
324     (*p).arena_index(0xff);
325     arena[i].marked_node = (N*) 0;
326     arena[i].previous = 0xff;
327     arena[i].next = 0xff;
328 }
329
330 N* front(header_type const & header, rank_type h, arena_type const &) const {
331     return header[h];
332 }
333
334 N* back(header_type const & header, rank_type h, arena_type const & a) const
335 {
336     if (header[h] == 0) {
337         return (N*) 0;
338     }
339     index_type i = (*header[h]).arena_index();
340     index_type k = a[i].previous;
341     return a[k].marked_node;
342 }
343
344 void push_front(header_type & header, N* p, arena_type & arena) {
345     rank_type h = (*p).rank();
346     index_type j = (*p).arena_index();
347     arena[j].marked_node = p;
348     if (header[h] == 0) {
349         arena[j].previous = j;
350         arena[j].next = j;
351     }
352     else {
353         index_type k = (*header[h]).arena_index();
354         index_type i = arena[k].previous;
355         arena[i].next = j;
356         arena[j].previous = i;
357         arena[j].next = k;
358         arena[k].previous = j;
359     }
360     header[h] = p;
361 }
362
363 void push_back(header_type & header, N* p, arena_type & arena) {
364     rank_type h = (*p).rank();
365     N* earlier_front = header[h];
366     push_front(header, p, arena);
367     if (earlier_front != 0) {
368         header[h] = earlier_front;
369     }
370 }
371
372 void erase(header_type & header, N* q, arena_type & arena) {
373     rank_type h = (*q).rank();
374     index_type j = (*q).arena_index();
375     index_type i = arena[j].previous;
376     index_type k = arena[j].next;

```

```

376     arena[j].marked_node = (N*) 0;
377     arena[j].previous = 0xff;
378     arena[j].next = 0xff;
379     if (i == j) {
380         header[h] = (N*) 0;
381     }
382     else {
383         arena[i].next = k;
384         arena[k].previous = i;
385         if (header[h] == q) {
386             header[h] = arena[k].marked_node;
387         }
388     }
389 }
390
391 bool get(bitvector_type w, size_type i) const {
392     w = w >> i;
393     return w & 1;
394 }
395
396 size_type most_significant_one(bitvector_type w) const {
397     return word_size - __builtin_clzl(w) - 1; // GNU gcc specific
398 }
399
400 bitvector_type set(bitvector_type w, size_type i) const {
401     if (get(w, i) == 0) {
402         w += (1ULL << i);
403     }
404     return w;
405 }
406
407 bitvector_type unset(bitvector_type w, size_type i) const {
408     if (get(w, i) == 1) {
409         w -= (1ULL << i);
410     }
411     return w;
412 }
413 };
414 }
415
416 #endif

```

Weak queue

§ 10 *heap-node.hpp*

```

1  /*
2   * This node can be used in a weak queue
3   *
4   * Author: Jyrki Katajainen © 2010–2011
5   */
6
7 #ifndef __CPHSTL_HEAP_NODE__
8 #define __CPHSTL_HEAP_NODE__
9
10 #include <algorithm> // std::swap
11
12 #include <cstddef> // std::size_t
13 #include <list>
14
15 namespace cphstl {
16     template <typename E>
17     class heap_node {
18     public:

```

```

19
20     typedef E element_type;
21     typedef unsigned char rank_type;
22     typedef std::size_t size_type;
23
24     heap_node* parent_;
25     heap_node* left_child_;
26     heap_node* right_child_;
27     E element_;
28
29 private:
30
31     heap_node();
32     heap_node(heap_node const &);
33     heap_node & operator=(heap_node const &);
34
35 public:
36
37     template <typename A>
38     heap_node(E const & v, A const & = A())
39         : parent_(0), left_child_(0), right_child_(0), element_(v) {
40     }
41
42     rank_type rank() const {
43         heap_node const* p = this;
44         rank_type h = 0;
45         while ((*p).right() != 0) {
46             p = (*p).right();
47             h += 1;
48         }
49         return h;
50     }
51
52     bool is_root() const {
53         return parent_ == 0;
54     }
55
56     bool is_leaf() const {
57         return right_child_ == 0;
58     }
59
60     bool is_marked() const {
61         return false;
62     }
63
64     heap_node* parent() const {
65         return parent_;
66     }
67
68     heap_node* left() const {
69         return left_child_;
70     }
71
72     heap_node* right() const {
73         return right_child_;
74     }
75
76     E const & element() const {
77         return element_;
78     }
79
80     void rank(rank_type) {
81     }
82
83     void up(rank_type delta = 1) {

```

```

84     }
85
86     void parent(heap_node* p) {
87         parent_ = p;
88     }
89
90     void left(heap_node* p) {
91         left_child_ = p;
92     }
93
94     void right(heap_node* p) {
95         right_child_ = p;
96     }
97
98     E& element() {
99         return element_;
100    }
101
102    void make_singleton() {
103        heap_node* q = this;
104        (*q).parent(0);
105        (*q).left(0);
106        (*q).right(0);
107    }
108
109    heap_node* distinguished_ancestor() const {
110        heap_node const* q = this;
111        heap_node* p = (*q).parent();
112        while (p != 0 && (*p).left() == q) {
113            q = p;
114            p = (*p).parent();
115        }
116        return p;
117    }
118
119 };
120 }
121
122 #endif

```

§ 11 *root-registry.hpp*

```

1  /*
2  2  A root registry keeps track of the roots of the perfect weak heaps
3  3  in a weak queue. The interface is almost identical to that of a
4  4  level registry. In this particular implementation the regular
5  5  numeral system is used to maintain the sequence of digits denoting
6  6  the cardinality of heaps of some particular rank. This registry is
7  7  a duplicate registry with the following characteristics:
8  8
9  9  – key: rank stored at the nodes or computed on the fly
10 10 – duplicate linking: none since there are at most two duplicates per key
11 11 – access: a root of the given rank; the smallest rank having duplicates
12 12 – duplicate reduction: obey the rules of the numeral system
13 13
14 14 The digit sequence must be kept in the form (0 | 1 | 01*2)*.
15 15
16 16 Author: Jyrki Katajainen © 2010–2011
17 17 */
18 18
19 19 #ifndef __CPHSTL_ROOT_REGISTRY__
20 20 #define __CPHSTL_ROOT_REGISTRY__
21 21
22 22 #include <algorithm> // std::swap

```

```

23
24 #include <climits> // CHAR_BIT
25 #include "comparator-proxy.h++"
26 #include <cstdint> // std::size_t
27 #include <utility> // std::pair
28 #include <vector>
29
30 namespace cphstl {
31 template<typename N, typename M, typename C>
32 class root_registry {
33 public:
34
35     typedef N node_type;
36     typedef M modifier_type;
37     typedef C comparator_type;
38     typedef unsigned char rank_type;
39     typedef std::size_t size_type;
40
41 protected:
42
43     typedef unsigned long W;
44     enum {word_size = CHAR_BIT * sizeof(W)};
45
46     comparator_proxy<C> comparator;
47     M modifier;
48     W occupied;
49     W saturated;
50     std::vector<std::pair<N*, N*> > roots;
51
52 public:
53
54     root_registry(C const & c = C())
55         : comparator(c), modifier(c), occupied(W(0)), saturated(W(0)), roots() {
56         typedef std::pair<N*, N*> pair_type;
57         pair_type default_pair((N*) 0, (N*) 0);
58         roots.resize(word_size, default_pair);
59     }
60
61     ~root_registry() {
62     }
63
64     bool empty() const {
65         return occupied == W(0);
66     }
67
68     N* __first__() const {
69         if (empty()) {
70             return (N*) 0;
71         }
72         size_type i = least_significant_one(occupied);
73         return (N*) roots[i].first;
74     }
75
76     N* top() const {
77         if (empty()) {
78             return 0;
79         }
80         N* max = 0;
81         size_type i = least_significant_one(occupied);
82         size_type k = most_significant_one(occupied);
83         for (size_type j = i; j <= k; ++j) {
84             N* p = roots[j].first;
85             if (p == 0) {
86                 continue;
87             }

```

```

88     if (max == 0 || comparator((*max).element(), (*p).element())) {
89         max = p;
90     }
91     N* q = roots[j].second;
92     if (q != 0 && comparator((*max).element(), (*q).element())) {
93         max = q;
94     }
95     }
96     return max;
97 }
98
99 void insert(N* p) {
100     // Assumption: An insertion in the middle is always a replacement
101     if (! (*p).is_root()) {
102         return;
103     }
104     rank_type h = (*p).rank();
105     if (get(occupied, h)) {
106         saturated = set(saturated, h);
107         roots[h].second = p;
108     }
109     else {
110         occupied = set(occupied, h);
111         roots[h].first = p;
112     }
113 }
114
115 void extract(N* p) {
116     if (! (*p).is_root()) {
117         return;
118     }
119     rank_type const h = (*p).rank();
120     if (get(saturated, h)) {
121         if (roots[h].second == p) {
122             saturated = unset(saturated, h);
123             roots[h].second = 0;
124         }
125         else if (roots[h].first == p) {
126             roots[h].first = roots[h].second;
127             roots[h].second = 0;
128             saturated = unset(saturated, h);
129         }
130     }
131     else {
132     }
133     else {
134         occupied = unset(occupied, h);
135         roots[h].first = 0;
136     }
137 }
138
139 void reduce() {
140     if (saturated == W(0)) {
141         return;
142     }
143     size_type h = least_significant_one(saturated);
144     N* q = roots[h].first;
145     N* r = roots[h].second;
146     extract(q);
147     extract(r);
148     N* p = modifier.join(q, r, *this);
149     if (get(occupied, h + 1)) {
150         saturated = set(saturated, h + 1);
151         roots[h + 1].second = p;
152     }

```

```

153     else {
154         occupied = set(occupied, h + 1);
155         roots[h + 1].first = p;
156     }
157 }
158
159 void swap(root_registry& other) {
160     std::swap(comparator, other.comparator);
161     std::swap(modifier, other.modifier);
162     std::swap(occupied, other.occupied);
163     std::swap(saturated, other.saturated);
164     std::swap(roots, other.roots);
165 }
166
167 protected:
168
169     bool get(W word, size_type i) const {
170         word = word >> i;
171         return word & 1;
172     }
173
174     size_type least_significant_one(W word) const {
175         return __builtin_ctzl(word); // GNU gcc specific
176     }
177
178     size_type most_significant_one(W word) const {
179         return word_size - __builtin_clzl(word) - 1; // GNU gcc specific
180     }
181
182     W set(W word, size_type i) const {
183         // Assumption: the bit at position i is 0
184         word += (1 << i);
185         return word;
186     }
187
188     W unset(W word, size_type i) const {
189         // Assumption: the bit at position i is 1
190         word -= (1 << i);
191         return word;
192     }
193 };
194 }
195
196 #endif

```

§ 12 *relaxed-weak-queue.h++*

```

1 /*
2  A weak-queue framework which can be used to implement, for
3  example, weak queue, run-relaxed weak queue, and fat heap.
4
5  Operations to be supported by the root registry: default
6  constructor, destructor, empty, --first--, top, insert, extract,
7  reduce, swap, show, is_valid.
8
9  Operations to be supported by the mark registry: default
10 constructor, destructor, empty, --first--, top, insert, extract,
11 reduce, swap, show, is_valid.
12
13 Author: Jyrki Katajainen © 2010–2011
14 */
15
16 #ifndef __CPHSTL_RELAXED_WEAK_QUEUE__
17 #define __CPHSTL_RELAXED_WEAK_QUEUE__

```

```

18
19 #include <algorithm>
20
21 #include "comparator-proxy.h++"
22 #include <cstdint> // std::size_t
23 #include <utility> // std::pair
24 #include <vector>
25
26 namespace cphstl {
27     template <typename E, typename C, typename N, typename M, typename R,
28             typename V>
29     class relaxed_weak_queue {
30     public:
31
32         // types
33
34         typedef E element_type;
35         typedef C comparator_type;
36         typedef N node_type;
37         typedef M modifier_type;
38         typedef R root_registry_type;
39         typedef V mark_registry_type;
40         typedef E& reference;
41         typedef E const& const_reference;
42         typedef std::size_t size_type;
43
44     protected:
45
46         // variables
47
48         comparator_proxy<C> comparator_;
49         M modifier;
50         R root_registry;
51         V mark_registry;
52         size_type n;
53
54     public:
55
56         // structors
57
58         explicit relaxed_weak_queue(C const& c = C())
59             : comparator_(c), modifier(c), root_registry(c), mark_registry(c),
60             n(0) {
61         }
62
63         ~relaxed_weak_queue() {
64         }
65
66         // iterators
67
68         N* begin() {
69             if (root_registry.empty()) {
70                 return (N*) 0;
71             }
72             return root_registry.__first__();
73         }
74
75         N const* begin() const {
76             if (root_registry.empty()) {
77                 return (N*) 0;
78             }
79             return root_registry.__first__();
80         }
81
82         N* end() {

```

```

83     return (N*) 0;
84 }
85
86 N const* end() const {
87     return (N*) 0;
88 }
89
90 // accessors
91
92 C get_comparator_() const {
93     return C(comparator_.subject());
94 }
95
96 C comparator() const {
97     return C(comparator_.subject());
98 }
99
100 size_type size() const {
101     return n;
102 }
103
104 size_type max_size() const {
105     typename std::vector<int>::allocator_type a;
106     size_type available_memory = a.max_size() * sizeof(int); // in bytes
107     return n + available_memory / sizeof(N);
108 }
109
110 N* top() const {
111     N* p = root_registry.top();
112     N* q = mark_registry.top();
113     if (p == 0) {
114         return q;
115     }
116     if (q == 0 || comparator_((*q).element(), (*p).element())) {
117         return p;
118     }
119     return q;
120 }
121
122 // modifiers
123
124 N* insert(N* p) {
125     (*p).make_singleton();
126     root_registry.insert(p);
127     root_registry.reduce();
128     ++n;
129     return p;
130 }
131
132 N* extract() {
133     N* root = root_registry._first_();
134     root_registry.extract(root);
135     N* q = (*root).right();
136     size_type removed_markings = 0;
137     while (q != 0) {
138         if ((*q).is_marked()) {
139             mark_registry.extract(q);
140             ++removed_markings;
141         }
142         N* r = (*q).left();
143         (*q).parent(0);
144         (*q).left(0);
145         root_registry.insert(q);
146         q = r;
147     }

```

```

148     (*root).make_singleton();
149     --n;
150     if (removed_markings == 0) {
151         mark_registry.reduce(root_registry);
152     }
153     (*root).make_singleton();
154     return root;
155 }
156
157 N* extract(N* p) {
158     N* replacement = extract();
159     if (p == replacement) {
160         (*p).make_singleton();
161         return p;
162     }
163     root_registry.extract(p);
164     size_type removed_markings = 0;
165     if ((*p).is_marked()) {
166         mark_registry.extract(p);
167         ++removed_markings;
168     }
169     N* r = p;
170     N* s = (*p).right();
171     while (s != 0) {
172         if ((*s).is_marked()) {
173             mark_registry.extract(s);
174             ++removed_markings;
175         }
176         r = s;
177         s = (*r).left();
178     }
179     N* q = replacement;
180     s = r;
181     while (s != p) {
182         r = (*s).parent();
183         (*s).left(0);
184         q = modifier.join(q, s, root_registry);
185         s = r;
186     }
187     modifier.replace(p, q);
188     root_registry.insert(q);
189     mark_registry.insert(q, root_registry);
190     if (removed_markings == 0) {
191         mark_registry.reduce(root_registry);
192     }
193     (*p).make_singleton();
194     return p;
195 }
196
197 void increase(N* p, E const& v) {
198     (*p).element() = v;
199     mark_registry.insert(p, root_registry);
200     mark_registry.reduce(root_registry);
201 }
202
203 void meld(relaxed_weak_queue& other) {
204     if (this == &other) {
205         return;
206     }
207     if (size() < other.size()) {
208         swap(other);
209     }
210     while (! other.root_registry.empty()) {
211         N* t = other.root_registry.__first__();
212         other.root_registry.extract(t);

```

```

213     root_registry.insert(t);
214     root_registry.reduce();
215 }
216 while (! other.mark_registry.empty()) {
217     N* v = other.mark_registry.__first__();
218     other.mark_registry.extract(v);
219     mark_registry.insert(v, root_registry);
220     mark_registry.reduce();
221 }
222 n += other.n;
223 other.n = 0;
224 }
225
226 void swap(relaxed_weak_queue & other) {
227     std::swap(comparator_, other.comparator_);
228     root_registry.swap(other.root_registry);
229     mark_registry.swap(other.mark_registry);
230     std::swap(n, other.n);
231 }
232
233 };
234 }
235
236 #endif

```

§ 13 *weak-queue.i++*

```

1 #include "heap-node.h++"
2 #include "meldable-priority-queue.h++"
3 #include <memory>
4 #include "naive-mark-registry.h++"
5 #include "priority-queue-iterator.h++"
6 #include "relaxed-heap-modifier.h++"
7 #include "relaxed-weak-queue.h++"
8 #include "root-registry.h++"
9
10 long long comps = 0;
11
12 template <typename T>
13 class counting_comparator {
14 public:
15
16     bool operator()(T const & a, T const & b) const {
17         ++comps;
18         return a < b;
19     }
20 };
21
22 typedef long long E;
23 typedef counting_comparator<E> C;
24 typedef std::allocator<E> A;
25
26 typedef cphstl::heap_node<E> N;
27 typedef cphstl::relaxed_heap_modifier<N, C> M;
28 typedef cphstl::root_registry<N, M, C> R;
29 typedef cphstl::naive_mark_registry<N, M, C> V;
30 typedef cphstl::relaxed_weak_queue<E, C, N, M, R, V> W;
31
32 typedef cphstl::priority_queue_iterator<N, W> I;
33 typedef cphstl::priority_queue_iterator<N, W, true> J;
34 typedef cphstl::meldable_priority_queue<E, C, A, N, W, I, J> Q;

```

Relaxed weak queue

§ 14 *perfect-weak-heap-node.h++*

```

1  /*
2  This node can be used in a weak queue and a rank-relaxed weak queue
3
4  Author: Jyrki Katajainen © 2010–2011
5  */
6
7  #ifndef __CPHSTL_PERFECT_WEAK_HEAP_NODE__
8  #define __CPHSTL_PERFECT_WEAK_HEAP_NODE__
9
10 #include <algorithm> // std::swap
11
12 #include <climits> // CHAR_BIT
13 #include <list>
14
15 namespace cphstl {
16     template <typename E>
17     class perfect_weak_heap_node {
18     public:
19
20         typedef E element_type;
21         typedef unsigned char rank_type;
22         typedef unsigned char index_type;
23
24     protected:
25
26         rank_type rank_;
27         index_type arena_index_;
28         perfect_weak_heap_node* parent_;
29         perfect_weak_heap_node* left_child_;
30         perfect_weak_heap_node* right_child_;
31         E element_;
32
33     public:
34
35         template <typename A>
36         perfect_weak_heap_node(E const & v, A const & = A())
37             : rank_(0), arena_index_(0xff), parent_(0), left_child_(0),
38               right_child_(0), element_(v) {
39         }
40
41         rank_type rank() const {
42             return rank_;
43         }
44
45         bool is_root() const {
46             return parent_ == 0;
47         }
48
49         bool is_marked() const {
50             return arena_index_ != 0xff;
51         }
52
53         index_type arena_index() const {
54             return arena_index_;
55         }
56
57         perfect_weak_heap_node* parent() const {
58             return parent_;
59         }
60

```

```

61     perfect_weak_heap_node* left() const {
62         return left_child_;
63     }
64
65     perfect_weak_heap_node* right() const {
66         return right_child_;
67     }
68
69     E const& element() const {
70         return element_;
71     }
72
73     void rank(rank_type r) {
74         rank_ = r;
75     }
76
77     void down(rank_type delta = 1) {
78         rank_ -= delta;
79     }
80
81     void up(rank_type delta = 1) {
82         rank_ += delta;
83     }
84
85     void arena_index(index_type i) {
86         arena_index_ = i;
87     }
88
89     void parent(perfect_weak_heap_node* p) {
90         parent_ = p;
91     }
92
93
94     void left(perfect_weak_heap_node* p) {
95         left_child_ = p;
96     }
97
98
99     void right(perfect_weak_heap_node* p) {
100        right_child_ = p;
101    }
102
103
104     E& element() {
105         return element_;
106     }
107
108     void make_singleton() {
109         perfect_weak_heap_node* q = this;
110         (*q).rank(0);
111         (*q).arena_index(0xff);
112         (*q).parent(0);
113         (*q).left(0);
114         (*q).right(0);
115     }
116
117 };
118 }
119
120 #endif

```

§ 15 *rank-relaxed-weak-queue.i++*

```
1 #include "eager-mark-registry.h++"
```

```

2 #include "meldable-priority-queue.h++"
3 #include <memory>
4 #include "perfect-weak-heap-node.h++"
5 #include "priority-queue-iterator.h++"
6 #include "relaxed-heap-modifier.h++"
7 #include "relaxed-weak-queue.h++"
8 #include "root-registry.h++"
9
10 long long comps = 0;
11
12 template <typename T>
13 class counting_comparator {
14 public:
15
16     bool operator()(T const & a, T const & b) const {
17         ++comps;
18         return a < b;
19     }
20 };
21
22 typedef long long E;
23 typedef counting_comparator<E> C;
24 typedef std::allocator<E> A;
25
26 typedef cphstl::perfect_weak_heap_node<E> N;
27 typedef cphstl::relaxed_heap_modifier<N, C> M;
28 typedef cphstl::root_registry<N, M, C> R;
29 typedef cphstl::eager_mark_registry<N, M, C> V;
30 typedef cphstl::relaxed_weak_queue<E, C, N, M, R, V> W;
31
32 typedef cphstl::priority_queue_iterator<N, W> I;
33 typedef cphstl::priority_queue_iterator<N, W, true> J;
34 typedef cphstl::meldable_priority_queue<E, C, A, N, W, I, J> Q;

```

§ 16 *run-relaxed-weak-queue.i++*

```

1 #include "lazy-mark-registry.h++"
2 #include "meldable-priority-queue.h++"
3 #include <memory>
4 #include "perfect-weak-heap-node.h++"
5 #include "priority-queue-iterator.h++"
6 #include "relaxed-heap-modifier.h++"
7 #include "relaxed-weak-queue.h++"
8 #include "root-registry.h++"
9
10 long long comps = 0;
11
12 template <typename T>
13 class counting_comparator {
14 public:
15
16     bool operator()(T const & a, T const & b) const {
17         ++comps;
18         return a < b;
19     }
20 };
21
22 typedef long long E;
23 typedef counting_comparator<E> C;
24 typedef std::allocator<E> A;
25
26 typedef cphstl::perfect_weak_heap_node<E> N;
27 typedef cphstl::relaxed_heap_modifier<N, C> M;
28 typedef cphstl::root_registry<N, M, C> R;

```

```

29 typedef cphstl::lazy_mark_registry<N, M, C> V;
30 typedef cphstl::relaxed_weak_queue<E, C, N, M, R, V> W;
31
32 typedef cphstl::priority_queue_iterator<N, W> I;
33 typedef cphstl::priority_queue_iterator<N, W, true> J;
34 typedef cphstl::meldable_priority_queue<E, C, A, N, W, I, J> Q;

```

Weak heap

§ 17 *level-registry.hpp*

```

1  /*
2   A level registry keeps track of the nodes stored at some levels of a
3   tree. This registry is a duplicate registry where nodes with some
4   key values are considered uninteresting.
5
6   - key: depth stored at the nodes
7   - duplicate linking: previous and next pointers stored at the nodes
8   - access: the root and an arbitrary leaf at the given depth
9   - duplicate reduction: none
10
11  Author: Jyrki Katajainen © 2011
12 */
13
14 #ifndef __CPHSTL_LEVEL_REGISTRY__
15 #define __CPHSTL_LEVEL_REGISTRY__
16
17 #include <algorithm> // std::swap
18
19 #include <cstdint> // std::size_t
20
21 namespace cphstl {
22     template <typename N>
23     class level_registry {
24     public:
25
26         typedef N node_type;
27         typedef unsigned char rank_type;
28         typedef std::size_t size_type;
29
30     protected:
31
32         rank_type depth_;
33         N* root_;
34         N* second_last_level;
35         N* last_level;
36
37     public:
38
39         level_registry()
40             : depth_(0), root_(0), second_last_level(0), last_level(0) {}
41
42         ~level_registry() {}
43
44         rank_type depth() const {
45             return depth_;
46         }
47
48         N* root() const {
49             return (N*) root_;
50         }
51
52     };
53

```

```

54     template <typename integer>
55     N* leaf(integer d) const {
56         if (d == integer(depth() - 1)) {
57             if (second_last_level != 0) {
58                 return (N*) second_last_level;
59             }
60         }
61         if (d == integer(depth())) {
62             if (last_level != 0) {
63                 return (N*) (*last_level).previous();
64             }
65         }
66         return (N*) root_;
67     }
68
69     template <typename integer>
70     void depth(integer delta) {
71         if (delta == integer(+1)) {
72             if (root_ == 0) {
73                 last_level = 0;
74                 depth_ = 0;
75             }
76             else {
77                 second_last_level = last_level;
78                 last_level = 0;
79                 depth_ += 1;
80             }
81         }
82         else if (delta == integer(-1)) {
83             if (depth_ == 0) {
84                 root_ = 0;
85                 second_last_level = 0;
86             }
87             else {
88                 last_level = second_last_level;
89                 second_last_level = 0;
90                 depth_ -= 1;
91             }
92         }
93     }
94
95     void insert(N* p) {
96         if ((*p).depth() == 0) {
97             root_ = p;
98             return;
99         }
100        if ((*p).depth() == depth() - 1) {
101            if (depth() == 1) {
102                root_ = p;
103                second_last_level = 0;
104            }
105            else if ((*p).degree() != 2) {
106                second_last_level = circular_list_insert(second_last_level, p);
107            }
108            return;
109        }
110        if ((*p).depth() == depth()) {
111            last_level = circular_list_insert(last_level, p);
112            return;
113        }
114    }
115
116     void extract(N* q) {
117         if (root_ == q) {
118             root_ = 0;

```

```

119     return;
120 }
121 if (second_last_level == q) {
122     second_last_level = circular_list_extract(second_last_level, q);
123     return;
124 }
125 if (last_level == q) {
126     last_level = circular_list_extract(last_level, q);
127     return;
128 }
129 N* p = (*q).previous();
130 N* r = (*q).next();
131 if (p == 0) {
132     return;
133 }
134 (*p).next(r);
135 (*r).previous(p);
136 (*q).previous(0);
137 (*q).next(0);
138 }
139
140 void swap(level_registry & other) {
141     std::swap(depth_, other.depth_);
142     std::swap(root_, other.root_);
143     std::swap(second_last_level, other.second_last_level);
144     std::swap(last_level, other.last_level);
145 }
146
147 protected:
148
149 N* circular_list_insert(N* handle, N* q) {
150     if (handle == 0) {
151         (*q).next(q);
152         (*q).previous(q);
153         return q;
154     }
155     N* p = (*handle).previous();
156     (*p).next(q);
157     (*q).previous(p);
158     (*q).next(handle);
159     (*handle).previous(q);
160     return handle;
161 }
162
163 N* circular_list_extract(N* handle, N* q) {
164     N* p = (*q).previous();
165     N* r = (*q).next();
166     (*q).previous(0);
167     (*q).next(0);
168     if (p == q) {
169         return (N*) 0;
170     }
171     (*p).next(r);
172     (*r).previous(p);
173     if (handle == q) {
174         return r;
175     }
176     return handle;
177 }
178 };
179 }
180
181 #endif

```

§ 18 *weak-heap.i++*

```

1 #include "level-registry.h++"
2 #include "meldable-priority-queue.h++"
3 #include <memory>
4 #include "naive-mark-registry.h++"
5 #include "pointer-based-weak-heap.h++"
6 #include "priority-queue-iterator.h++"
7 #include "relaxed-heap-modifier.h++"
8 #include "weak-heap-node.h++"
9
10 long long comps = 0;
11
12 template <typename T>
13 class counting_comparator {
14 public:
15
16     bool operator()(T const & a, T const & b) const {
17         ++comps;
18         return a < b;
19     }
20 };
21
22 typedef long long E;
23 typedef counting_comparator<E> C;
24 typedef std::allocator<E> A;
25
26 typedef cphstl::weak_heap_node<E> N;
27 typedef cphstl::relaxed_heap_modifier<N, C> M;
28 typedef cphstl::level_registry<N> L;
29 typedef cphstl::naive_mark_registry<N, M, C> V;
30 typedef cphstl::pointer_based_weak_heap<E, C, N, M, L, V> W;
31
32 typedef cphstl::priority_queue_iterator<N, W> I;
33 typedef cphstl::priority_queue_iterator<N, W, true> J;
34 typedef cphstl::meldable_priority_queue<E, C, A, N, W, I, J> Q;

```

Relaxed weak heap

§ 19 *weak-heap-node.h++*

```

1 /*
2  A node used in a weak heap
3
4  Author: Jyrki Katajainen © 2009–2011
5 */
6
7 #ifndef __CPHSTL_WEAK_HEAP_NODE__
8 #define __CPHSTL_WEAK_HEAP_NODE__
9
10 #include <climits> // CHAR_BIT
11 #include <cstddef> // std::size_t
12 #include <list>
13
14 namespace cphstl {
15     template <typename E>
16     class weak_heap_node {
17     public:
18
19         typedef E element_type;
20         typedef unsigned char rank_type;
21         typedef unsigned char index_type;
22

```

```

23     rank_type depth_;
24     index_type arena_index_;
25     weak_heap_node* parent_;
26     weak_heap_node* left_;
27     weak_heap_node* right_;
28     weak_heap_node* previous_;
29     weak_heap_node* next_;
30     E element_;
31
32 protected:
33
34     typedef unsigned long W;
35     enum {word_size = CHAR_BIT * sizeof(W)};
36
37 public:
38
39     template <typename A>
40     weak_heap_node(E const & v, A const & = A())
41         : depth_(0), arena_index_(0xff), parent_(0), left_(0), right_(0),
42           previous_(0), next_(0), element_(v) {
43     }
44
45     bool is_root() const {
46         return parent_ == 0;
47     }
48
49     bool is_marked() const {
50         return arena_index_ != 0xff;
51     }
52
53     index_type degree() const {
54         return (left() != 0) + (right() != 0);
55     }
56
57     rank_type depth() const {
58         return depth_;
59     }
60
61     rank_type rank() const {
62         return word_size - depth_ - 1;
63     }
64
65     index_type arena_index() const {
66         return arena_index_;
67     }
68
69     weak_heap_node* parent() const {
70         return parent_;
71     }
72
73     weak_heap_node* left() const {
74         return left_;
75     }
76
77     weak_heap_node* right() const {
78         return right_;
79     }
80
81     weak_heap_node* previous() const {
82         return previous_;
83     }
84
85     weak_heap_node* next() const {
86         return next_;
87     }

```

```

88
89     E const & element() const {
90         return element_;
91     }
92
93     void depth(rank_type d) {
94         depth_ = d;
95     }
96
97     void rank(rank_type d) {
98         depth_ = word_size - d - 1;
99     }
100
101     void down(rank_type delta = 1) {
102         depth_ += delta;
103     }
104
105     void up(rank_type delta = 1) {
106         depth_ -= delta;
107     }
108
109     void arena_index(index_type i) {
110         arena_index_ = i;
111     }
112
113     void parent(weak_heap_node* v) {
114         parent_ = v;
115     }
116
117     void left(weak_heap_node* v) {
118         left_ = v;
119     }
120
121     void right(weak_heap_node* v) {
122         right_ = v;
123     }
124
125     void previous(weak_heap_node* v) {
126         previous_ = v;
127     }
128
129     void next(weak_heap_node* v) {
130         next_ = v;
131     }
132
133     E & element() {
134         return element_;
135     }
136
137     void make_singleton() {
138         weak_heap_node* q = this;
139         (*q).parent(0);
140         (*q).left(0);
141         (*q).right(0);
142         (*q).depth(0);
143     }
144
145     weak_heap_node* distinguished_ancestor() const {
146         weak_heap_node const* q = this;
147         weak_heap_node* p = (*q).parent();
148         while (p != 0 && (*p).left() == q) {
149             q = p;
150             p = (*p).parent();
151         }
152         return p;

```

```

153     }
154
155     };
156 }
157
158 #endif

```

§ 20 *pointer-based.weak-heap.h++*

```

1  /*
2  A pointer-based weak heap
3
4  A level registry has properties depth (get/set), root (get), and
5  leaf (get), and it has the member functions default constructor,
6  destructor, insert, extract, swap, show, is_valid.
7
8  A mark registry has the properties empty (get) and top (get),
9  and it has the member functions default constructor, destructor,
10 --first--, --next--, insert, extract, reduce, swap, show, is_valid.
11
12 Author: Jyrki Katajainen © 2010–2011
13 */
14
15 #ifndef __CPHSTL_POINTER_BASED_WEAK_HEAP__
16 #define __CPHSTL_POINTER_BASED_WEAK_HEAP__
17
18 #include <algorithm> // std::swap
19
20 #include <cmath> // ilogb
21 #include "comparator-proxy.h++"
22 #include <cstdlib> // std::size_t
23 #include <vector>
24
25 extern int ilogb(double) throw();
26
27 namespace cphstl {
28     template <typename E, typename C, typename N, typename M, typename L,
29             typename V>
30     class pointer_based_weak_heap {
31     public:
32
33         // types
34
35         typedef E element_type;
36         typedef C comparator_type;
37         typedef N node_type;
38         typedef M modifier_type;
39         typedef L level_registry_type;
40         typedef V mark_registry_type;
41         typedef unsigned char depth_type;
42         typedef E& reference;
43         typedef E const& const_reference;
44         typedef std::size_t size_type;
45
46     protected:
47
48         // variables
49
50         comparator_proxy<C> comparator;
51         M modifier;
52         L level_registry;
53         V mark_registry;
54         size_type n;
55

```

```

56 public:
57
58     // structors
59
60     explicit pointer_based_weak_heap(C const& c = C())
61         : comparator(c), modifier(c), level_registry(), mark_registry(c),
62           n(0) {
63     }
64
65     ~pointer_based_weak_heap() {
66     }
67
68     // iterators
69
70     N const* begin() const {
71         return level_registry.root();
72     }
73
74     N const* end() const {
75         return (N*) 0;
76     }
77
78     N* begin() {
79         return (N*) level_registry.root();
80     }
81
82     N* end() {
83         return (N*) 0;
84     }
85
86     // accessors
87
88     C get_comparator() const {
89         return C(comparator.subject());
90     }
91
92     size_type size() const {
93         return n;
94     }
95
96     size_type max_size() const {
97         typename std::vector<int>::allocator_type a;
98         size_type available_memory = a.max_size() * sizeof(int); // in bytes
99         return n + available_memory / sizeof(N);
100    }
101
102    N* top() const {
103        N* p = level_registry.root();
104        N* q = (N*) mark_registry.top();
105        if (p == 0 || q == 0 || comparator((*q).element(), (*p).element())) {
106            return p;
107        }
108        return q;
109    }
110
111    // modifiers
112
113    N* insert(N* q) {
114        if (n == 0) {
115            (*q).make_singleton();
116            level_registry.depth(+1);
117            level_registry.insert(q);
118        }
119        else {
120            if (power_of_two(n)) {

```

```

121         level_registry.depth(+1);
122     }
123     N* p = level_registry.leaf(level_registry.depth() - 1);
124     modifier.add_leaf(p, q, level_registry, mark_registry);
125     mark_registry.reduce(level_registry);
126 }
127 ++n;
128 return q;
129 }
130
131 N* extract() {
132     --n;
133     N* p;
134     if (n == 0) {
135         p = level_registry.root();
136         level_registry.extract(p);
137         level_registry.depth(-1);
138     }
139     else {
140         p = level_registry.leaf(level_registry.depth());
141         modifier.cut_leaf(p, level_registry, mark_registry);
142         if (power_of_two(n)) {
143             level_registry.depth(-1);
144         }
145         mark_registry.reduce(level_registry);
146     }
147     return p;
148 }
149
150 N* extract(N* p) {
151     if ((*p).depth() == level_registry.depth()) {
152         modifier.cut_leaf(p, level_registry, mark_registry);
153         --n;
154         if (n == 0 || power_of_two(n)) {
155             level_registry.depth(-1);
156         }
157         mark_registry.reduce(level_registry);
158         return p;
159     }
160     size_type removed_markings = 0;
161     if ((*p).is_marked()) {
162         ++removed_markings;
163         mark_registry.extract(p);
164     }
165     N* r = p;
166     N* s = (*p).right();
167     while (s != 0) {
168         if ((*s).is_marked()) {
169             ++removed_markings;
170             mark_registry.extract(s);
171         }
172         r = s;
173         s = (*r).left();
174     }
175     N* replacement;
176     if ((*r).depth() == level_registry.depth()) {
177         replacement = r;
178         r = (*r).parent();
179     }
180     else {
181         replacement = level_registry.leaf(level_registry.depth());
182     }
183     modifier.cut_leaf(replacement, level_registry, mark_registry);
184     N* q = replacement;
185     (*q).depth((*r).depth());

```

```

186     s = r;
187     while (s != p) {
188         r = (*s).parent();
189         (*s).left(0);
190         level_registry.extract(s);
191         q = modifier.join(q, s, level_registry);
192         s = r;
193     }
194     level_registry.extract(p);
195     modifier.replace(p, q);
196     level_registry.insert(q);
197     mark_registry.insert(q, level_registry);
198     if (removed_markings == 0) {
199         mark_registry.reduce(level_registry);
200     }
201     --n;
202     if (power_of_two(n)) {
203         level_registry.depth(-1);
204     }
205     return p;
206 }
207
208 void increase(N* p, E const& v) {
209     (*p).element() = v;
210     mark_registry.insert(p, level_registry);
211     mark_registry.reduce(level_registry);
212 }
213
214 void meld(pointer_based_weak_heap& other) {
215     if (this == &other) {
216         return;
217     }
218     if (size() < other.size()) {
219         swap(other);
220     }
221     while (other.size() != 0) {
222         insert(other.extract());
223     }
224 }
225
226 void swap(pointer_based_weak_heap& other) {
227     std::swap(comparator, other.comparator);
228     level_registry.swap(other.level_registry);
229     mark_registry.swap(other.mark_registry);
230     std::swap(n, other.n);
231 }
232
233 protected:
234
235     template <typename integer>
236     bool power_of_two(integer m) {
237         integer k = ilogb(m);
238         integer rest = m - (1 << k);
239         return rest == 0;
240     }
241 };
242 };
243 }
244
245 #endif

```

§ 21 *rank-relaxed-weak-heap.i++*

```
1 #include "eager-mark-registry.h++"
```

```

2 #include "level-registry.h++"
3 #include "meldable-priority-queue.h++"
4 #include <memory>
5 #include "pointer-based-weak-heap.h++"
6 #include "priority-queue-iterator.h++"
7 #include "relaxed-heap-modifier.h++"
8 #include "weak-heap-node.h++"
9
10 long long comps = 0;
11
12 template <typename T>
13 class counting_comparator {
14 public:
15
16     bool operator()(T const & a, T const & b) const {
17         ++comps;
18         return a < b;
19     }
20 };
21
22 typedef long long E;
23 typedef counting_comparator<E> C;
24 typedef std::allocator<E> A;
25
26 typedef cphstl::weak_heap_node<E> N;
27 typedef cphstl::relaxed_heap_modifier<N, C> M;
28 typedef cphstl::level_registry<N> L;
29 typedef cphstl::eager_mark_registry<N, M, C> V;
30 typedef cphstl::pointer_based_weak_heap<E, C, N, M, L, V> W;
31
32 typedef cphstl::priority_queue_iterator<N, W> I;
33 typedef cphstl::priority_queue_iterator<N, W, true> J;
34 typedef cphstl::meldable_priority_queue<E, C, A, N, W, I, J> Q;

```

§ 22 *run-relaxed-weak-heap.i++*

```

1 #include "lazy-mark-registry.h++"
2 #include "level-registry.h++"
3 #include "meldable-priority-queue.h++"
4 #include <memory>
5 #include "pointer-based-weak-heap.h++"
6 #include "priority-queue-iterator.h++"
7 #include "relaxed-heap-modifier.h++"
8 #include "weak-heap-node.h++"
9
10 long long comps = 0;
11
12 template <typename T>
13 class counting_comparator {
14 public:
15
16     bool operator()(T const & a, T const & b) const {
17         ++comps;
18         return a < b;
19     }
20 };
21
22 typedef long long E;
23 typedef counting_comparator<E> C;
24 typedef std::allocator<E> A;
25
26 typedef cphstl::weak_heap_node<E> N;
27 typedef cphstl::relaxed_heap_modifier<N, C> M;
28 typedef cphstl::level_registry<N> L;

```

```

29 typedef cphstl::lazy_mark_registry<N, M, C> V;
30 typedef cphstl::pointer_based_weak_heap<E, C, N, M, L, V> W;
31
32 typedef cphstl::priority_queue_iterator<N, W> I;
33 typedef cphstl::priority_queue_iterator<N, W, true> J;
34 typedef cphstl::meldable_priority_queue<E, C, A, N, W, I, J> Q;

```

Directed graphs

§ 23 *graph-vertex.h++*

```

1  /*
2  A vertex that can be used in a directed graph
3
4  Author: Jyrki Katajainen © 2011
5  */
6
7  #ifndef __CPHSTL_GRAPH_VERTEX__
8  #define __CPHSTL_GRAPH_VERTEX__
9
10 namespace cphstl {
11     template <typename E>
12     class graph_vertex
13     : public B {
14     public:
15
16         typedef B base_type;
17         typedef typename B::element_type distance_type;
18         enum state_type {scanned = 0, labelled, unlabelled};
19
20     protected:
21
22         void* edge_;
23         state_type state_;
24
25     public:
26
27         graph_vertex(distance_type const& d = distance_type(0))
28         : B(d), edge_(0), state_(unlabelled) {
29         }
30
31         graph_vertex& operator=(graph_vertex const& v) {
32             B::element() = v.distance();
33             edge_ = v.edge();
34             state_ = v.state();
35             return *this;
36         }
37
38         void* edge() const {
39             return edge_;
40         }
41
42         state_type state() const {
43             return state_;
44         }
45
46         bool is_scanned() const {
47             return state_ != scanned;
48         }
49
50         bool is_labelled() const {
51             return state_ != labelled;
52         }
53

```

```

54     bool is_unlabelled() const {
55         return state_ != unlabelled;
56     }
57
58     distance_type distance() const {
59         return B::element();
60     }
61
62     template <typename E>
63     void edge(E* e) {
64         edge_ = (void*) e;
65     }
66
67     void state(state_type s) {
68         state_ = s;
69     }
70
71     void distance(distance_type dist) {
72         B::element() = dist;
73     }
74 };
75 }
76
77 #endif

```

§ 24 *combined-node.h++*

```

1  /*
2   A node that is both in a directed graph and in a Fibonacci heap
3
4   Author: Jyrki Katajainen © 2011
5  */
6
7  #ifndef __CPHSTL_COMBINED_NODE__
8  #define __CPHSTL_COMBINED_NODE__
9
10 #include <list>
11
12 namespace cphstl {
13     template <typename E>
14     class combined_node {
15     public:
16
17         // types
18
19         typedef E element_type;
20         typedef unsigned char degree_type;
21         enum state_type {scanned = 0, labelled, unlabelled};
22
23     protected:
24
25         // variables
26
27         degree_type degree_;
28         bool mark_;
29         state_type state_;
30         void* edge_;
31         combined_node* left_sibling_;
32         combined_node* right_sibling_;
33         combined_node* parent_;
34         combined_node* child_;
35         E element_;
36
37     public:

```

```

38
39 // structors
40
41 combined_node(E const & v = E(0))
42 : degree_(0), mark_(false), state_(unlabelled), edge_(0),
43   left_sibling_(this), right_sibling_(this), parent_(0), child_(0),
44   element_(v) {
45 }
46
47 ~combined_node() {
48 }
49
50 combined_node & operator=(combined_node const & v) {
51   element() = v.distance();
52   edge_ = v.edge();
53   state_ = v.state();
54   return *this;
55 }
56
57 // accesors
58
59 degree_type degree() const {
60   return degree_;
61 }
62
63 bool is_marked() const {
64   return mark_ == true;
65 }
66
67 bool is_root() const {
68   return parent_ == 0;
69 }
70
71 state_type state() const {
72   return state_;
73 }
74
75 bool is_scanned() const {
76   return state_ == scanned;
77 }
78
79 bool is_labelled() const {
80   return state_ == labelled;
81 }
82
83 bool is_unlabelled() const {
84   return state_ == unlabelled;
85 }
86
87 void* edge() const {
88   return edge_;
89 }
90
91 combined_node* left_sibling() const {
92   return left_sibling_;
93 }
94
95 combined_node* right_sibling() const {
96   return right_sibling_;
97 }
98
99 combined_node* child() const {
100   return child_;
101 }
102

```

```

103     combined_node* parent() const {
104         return parent_;
105     }
106
107     E const& distance() const {
108         return element();
109     }
110
111     E const& element() const {
112         return element_;
113     }
114
115     combined_node const* successor() const {
116         combined_node const* t = this;
117         if ((*t).child() != 0) {
118             t = (*t).child();
119             combined_node const* f = t;
120             combined_node const* e = t;
121             t = (*t).left_sibling();
122             while (t != f) {
123                 e = t;
124                 t = (*t).left_sibling();
125             }
126             return e;
127         }
128         combined_node const* p = (*t).parent();
129         while (p != 0 && t == (*p).child()) {
130             t = p;
131             p = (*p).parent();
132         }
133         return p;
134     }
135
136     // modifiers
137
138     void degree(degree_type r) {
139         degree_ = r;
140     }
141
142     void is_marked(bool b) {
143         mark_ = b;
144     }
145
146     void state(state_type s) {
147         state_ = s;
148     }
149
150     template <typename N>
151     void edge(N* e) {
152         edge_ = (void*) e;
153     }
154
155     void left_sibling(combined_node* p) {
156         left_sibling_ = p;
157     }
158
159     void right_sibling(combined_node* p) {
160         right_sibling_ = p;
161     }
162
163     void child(combined_node* p) {
164         child_ = p;
165     }
166
167

```

```

168     }
169
170     void parent(combined_node* p) {
171         parent_ = p;
172     }
173
174
175     void distance(E const& dist) {
176         element() = dist;
177     }
178
179     E& element() {
180         return element_;
181     }
182
183     void catenate(combined_node* y) {
184         combined_node* q = this;
185         combined_node* p = (*q).left_sibling();
186         combined_node* x = (*y).left_sibling();
187         (*p).right_sibling(y);
188         (*y).left_sibling(p);
189         (*q).left_sibling(x);
190         (*x).right_sibling(q);
191     }
192
193     void cut() {
194         combined_node* q = this;
195         combined_node* u = (*q).parent();
196         combined_node* p = (*q).left_sibling();
197         combined_node* r = (*q).right_sibling();
198         (*p).right_sibling(r);
199         (*r).left_sibling(p);
200         (*q).parent(0);
201         (*q).left_sibling(q);
202         (*q).right_sibling(q);
203         (*q).is_marked(false);
204         if (u != 0) {
205             (*u).degree((*u).degree() - 1);
206             if ((*u).degree() == 0) {
207                 (*u).child((combined_node*) 0);
208             }
209             else if ((*u).child() == q) {
210                 (*u).child(r);
211             }
212         }
213     }
214
215     void join(combined_node* q) {
216         combined_node* p = this;
217         combined_node* c = (*p).child();
218         if (c != 0) {
219             (*c).catenate(q);
220         }
221         (*q).parent(p);
222         (*p).child(q);
223         (*p).degree((*p).degree() + 1);
224     }
225 };
226 }
227
228 #endif

```

```

1  /*
2  An edge that can be used in a directed graph
3
4  Author: Jyrki Katajainen © 2011
5  */
6
7  #ifndef __CPHSTL_GRAPH_EDGE__
8  #define __CPHSTL_GRAPH_EDGE__
9
10 namespace cphstl {
11     template <typename V, typename W>
12     class graph_edge {
13     public:
14
15         typedef V vertex_type;
16         typedef W weight_type;
17
18     protected:
19
20         V* source_;
21         V* target_;
22         W weight_;
23
24     public:
25
26         graph_edge(V* s = 0, V* t = 0, W const & w = W(0))
27             : source_(s), target_(t), weight_(w) {
28         }
29
30         graph_edge & operator=(graph_edge const & e) {
31             source_ = e.source();
32             target_ = e.target();
33             weight_ = e.weight();
34             return *this;
35         }
36
37         V* source() const {
38             return source_;
39         }
40
41         V* target() const {
42             return target_;
43         }
44
45         W weight() const {
46             return weight_;
47         }
48
49         void source(V* s) {
50             source_ = s;
51         }
52
53         void target(V* t) {
54             target_ = t;
55         }
56
57         void weight(W w) {
58             weight_ = w;
59         }
60     };
61 }
62
63 #endif

```

§ 26 *directed-graph.h++*

```

1  /*
2   A directed graph
3
4   Author: Jyrki Katajainen © 2011
5  */
6
7  #ifndef __CPHSTL_DIRECTED_GRAPH__
8  #define __CPHSTL_DIRECTED_GRAPH__
9
10 #include <cstdlib> // std::size_t
11 #include <cstdlib> // std::rand
12 #include <vector>
13
14 namespace cphstl {
15     template <typename V, typename E, typename W>
16     class directed_graph {
17     public:
18
19         typedef V vertex_type;
20         typedef E edge_type;
21         typedef W weight_type;
22         typedef std::size_t size_type;
23
24     protected:
25
26         std::vector<V> vertex_pool;
27         std::vector<E> edge_pool;
28
29     public:
30
31         directed_graph(size_type n, size_type m)
32             : vertex_pool(), edge_pool() {
33             vertex_pool.resize(n);
34             edge_pool.resize(m);
35         }
36
37         size_type number_of_vertices() const {
38             return vertex_pool.size();
39         }
40
41         size_type number_of_edges() const {
42             return edge_pool.size();
43         }
44
45         V* random_vertex() const {
46             V* u = (V*) (&vertex_pool[std::rand() % number_of_vertices()]);
47             return u;
48         }
49
50         template <typename integer>
51         V* vertex(integer i) const {
52             size_type j = size_type(i);
53             return (V*) &vertex_pool[j];
54         }
55
56         template <typename integer>
57         E* edge(integer k) const {
58             size_type j = size_type(k);
59             return (E*) &edge_pool[j];
60         }
61
62         template <typename integer>
63         void vertex(integer i, V const & v) {

```

```

64     size_type j = size_type(i);
65     V* u = &vertex_pool[j];
66     *u = v;
67 }
68
69 template <typename integer>
70 void edge(integer k, E const& e) {
71     size_type j = size_type(k);
72     E* p = &edge_pool[j];
73     *p = e;
74 }
75
76 };
77 }
78
79 #undef for_each_vertex
80 #define for_each_vertex(v, __graph)\
81     std::size_t __i = 0;\
82     for (v = (__i != __graph.number_of_vertices()) ? __graph.vertex(__i) : v; __i !
        = __graph.number_of_vertices(); ++__i, v = (__i != __graph.
        number_of_vertices()) ? __graph.vertex(__i) : v)
83
84 #undef for_each_edge
85 #define for_each_edge(e, __graph)\
86     std::size_t __j = 0;\
87     for (e = (__j != __graph.number_of_edges()) ? __graph.edge(__j) : e; __j !=
        __graph.number_of_edges(); ++__j, e = (__j != __graph.number_of_edges()) ?
        __graph.edge(__j) : e)
88
89 #undef for_each_adjacent_edge
90 #define for_each_adjacent_edge(e, v, __graph)\
91     std::size_t __k = __graph.number_of_vertices();\
92     std::size_t __l = (std::size_t) (v - __graph.vertex(0));\
93     decltype(e) __p = (decltype(e)) (*v).edge();\
94     decltype(v) __u = v;\
95     ++__u;\
96     ++__l;\
97     decltype(e) __q = (__k == __l) ? __graph.edge(__graph.number_of_edges()) : (
        decltype(e)) (*__u).edge();\
98     if (__p != __q)\
99         for (e = __p; e != __q; ++e)
100
101 #endif

```

Benchmark drivers

§ 27 *dijkstra.cpp*

```

1 /*
2  An implementation of Dijkstra's algorithm for finding the
3  shortest-path distances from the given source to all vertices.
4
5  Authors: Stefan Edelkamp, Jyrki Katajainen, Lasse Petersen © 2011–2012
6  */
7
8  #include <algorithm> // std::sort
9  #include <cmath> // std::sqrt
10 #include "combined-node.h++"
11 #include <cstdlib> // std::rand, std::srand
12 #include <ctime>
13 #include "directed-graph.h++"
14 #include "fibonacci-heap.h++"
15 #include "graph-edge.h++"
16 #include <vector>

```

```

17
18 #include <LEDA/graph/graph.h>
19 #include <LEDA/core/random_source.h>
20
21 #ifndef NODES
22 #define NODES 1000000
23 #endif
24
25 long long comps = 0;
26
27 template <typename T>
28 class counting_comparator {
29 public:
30
31     bool operator()(T const & a, T const & b) const {
32         ++comps;
33         return a > b;
34     }
35 };
36
37 template <typename G, typename integer>
38 typename G::vertex_type* chain_graph(G& graph, integer n) {
39     typename G::vertex_type v;
40     for (integer i = 0; i < n - 1; ++i) {
41         v.edge(graph.edge(i));
42         graph.vertex(i, v);
43         typename G::edge_type e;
44         e.weight(rand() % n);
45         e.source(graph.vertex(i));
46         e.target(graph.vertex(i + 1));
47         graph.edge(i, e);
48     }
49     v.edge(graph.edge(n - 1));
50     graph.vertex(n - 1, v);
51     return graph.vertex(0);
52 }
53
54 template <typename E>
55 class source_comparator {
56 public:
57
58     bool operator()(E const & a, E const & b) const {
59         return a.source() < b.source();
60     }
61 };
62
63 template <typename integer>
64 integer moderate_graph(integer n) {
65     integer c = 0;
66     while (n > 0) {
67         n = n/2;
68         ++c;
69     }
70     integer m = NODES * c;
71     return m;
72 }
73
74 template <typename integer>
75 integer dense_graph(integer n) {
76     integer m = (std::size_t) (sqrt(NODES) * sqrt(NODES) * sqrt(NODES));
77     return m;
78 }
79
80 template <typename integer>
81 integer sparse_graph(integer n) {

```

```

82  integer m = integer(4) * n;
83  return m;
84 }
85
86 template <typename G, typename integer>
87 typename G::vertex_type* generate_graph(G& graph, integer n, integer m) {
88     typedef typename G::vertex_type V;
89     typedef typename G::edge_type E;
90     typedef typename G::weight_type W;
91
92     leda::GRAPH<W, W> r;
93     leda::random_graph(r, (int) n, (int) m);
94     std::vector<E> tmp;
95     leda::node_array<std::size_t> copy(r);
96     leda::node v;
97     std::size_t i = 0;
98     forall_nodes(v, r) {
99         copy[v] = i;
100        ++i;
101    }
102    leda::edge e;
103    forall_edges(e, r) {
104        E a;
105        std::size_t i = copy[leda::source(e)];
106        std::size_t j = copy[leda::target(e)];
107        a.source(graph.vertex(i));
108        a.target(graph.vertex(j));
109        a.weight(W(std::rand() % m));
110        tmp.push_back(a);
111    }
112    std::sort(tmp.begin(), tmp.end(), source_comparator<E>());
113    for (std::size_t i = 0; i < m; ++i) {
114        graph.edge(i, tmp[i]);
115    }
116    std::size_t k = 0;
117    V* current = graph.vertex(0);
118    E* cursor = graph.edge(0);
119    (*current).edge(cursor);
120    for (std::size_t j = 1; j < n; ++j) {
121        while (k < m && (*cursor).source() == current) {
122            ++k;
123            ++cursor;
124        }
125        ++current;
126        (*current).edge(cursor);
127    }
128    return graph.random_vertex();
129 }
130
131 template <typename G, typename Q>
132 void dijkstra(G& graph, typename G::vertex_type* s) {
133     typedef typename G::vertex_type V;
134     typedef typename G::edge_type E;
135     typedef typename Q::comparator_type C;
136     typedef typename Q::element_type W;
137     V* v = 0;
138     for_each_vertex(v, graph) {
139         (*v).state(V::unlabelled);
140     }
141     (*s).distance(W(0));
142     (*s).state(V::labelled);
143     Q heap;
144     C comparator = heap.comparator();
145     heap.insert(s);
146     while (heap.size() != 0) {

```

```

147     V* t = heap.top();
148     (*t).state(V::scanned);
149     W d = (*t).distance();
150     heap.extract(t);
151     E* e = 0;
152     for_each_adjacent_edge(e, t, graph) {
153         V* u = (*e).target();
154         W c = d + (*e).weight();
155         if ((*u).is_scanned()) {
156             ;
157         }
158         else if ((*u).is_unlabelled()) {
159             (*u).distance(c);
160             heap.insert(u);
161             (*u).state(V::labelled);
162         }
163         else if (comparator((*u).distance(), c)) {
164             (*u).distance(c);
165             heap.increase(u, c);
166         }
167         else {
168             ;
169         }
170     }
171 }
172 }
173
174 bool small_test() {
175     typedef long long W;
176     std::size_t n = 10;
177     typedef cphstl::combined_node<W> V;
178     typedef counting_comparator<W> C;
179     typedef cphstl::fibonacci_heap<W, C, V> R;
180     typedef cphstl::graph_edge<V, W> E;
181     typedef cphstl::directed_graph<V, E, W> G;
182     G g(n, n - 1);
183     V* s = chain_graph(g, n);
184
185     dijkstra<G, R>(g, s);
186     W sum = W(0);
187     typename G::edge_type* e = 0;
188     for_each_edge(e, g) {
189         sum += (*e).weight();
190     }
191     return true;
192 }
193
194 int main() {
195     typedef double W;
196     typedef counting_comparator<W> C;
197     typedef cphstl::combined_node<W> V;
198     typedef cphstl::fibonacci_heap<W, C, V> R;
199     typedef cphstl::graph_edge<V, W> E;
200     typedef cphstl::directed_graph<V, E, W> G;
201     std::size_t const n = NODES;
202     std::size_t const m = sparse_graph(n);
203
204     G graph(n, m);
205     V* s = generate_graph(graph, n, m);
206
207     comps = 0;
208     double running_time = 0.0;
209     int repetitions = 0;
210     while (running_time < 10.0) {
211         ++repetitions;

```

```

212     std::clock_t start = std::clock();
213     dijkstra<G, R>(graph, s);
214     std::clock_t stop = std::clock();
215     running_time += double(stop - start) / double(CLOCKS_PER_SEC);
216 }
217 long int lg_n = (long int) std::log10 (double(n)) / std::log10(2.0);
218 double time_per_run = running_time / double(repetitions);
219 double comps_per_run = double(comps) / double(repetitions);
220 std::cout << n << " " << time_per_run << " " << comps_per_run << "\n";
221 double scaled_time = time_per_run / double(n * lg_n + m);
222 double scaled_comps = comps_per_run / double(n * lg_n + m);
223 std::cout << n << " " << scaled_time << " " << scaled_comps << "\n";
224
225     return 0;
226 }

```

§ 28 *push-time.cpp*

```

1  #include <algorithm>
2  #include <cstdlib> // std::rand, std::srand
3  #include <ctime>
4
5  #ifndef NUMBER
6  #define NUMBER 1000000
7  #endif
8
9  #ifndef DATA
10 #define DATA increasing_sequence
11 #endif
12
13 #include "data-structure.i++" // Q comes from here
14 #include <iterator> // defines std::iterator_traits
15
16 template <typename I>
17 void increasing_sequence(I first, I past_the_end) {
18     typedef typename std::iterator_traits<I>::value_type V;
19     I q;
20     for (q = first; q != past_the_end; ++q) {
21         *q = V(unsigned(q - first));
22     }
23 }
24
25 template <typename I>
26 void decreasing_sequence(I first, I past_the_end) {
27     typedef typename std::iterator_traits<I>::value_type V;
28     I q;
29     for (q = first; q != past_the_end; ++q) {
30         *q = V(unsigned(past_the_end - q));
31     }
32 }
33
34 template <typename I>
35 void random_sequence(I first, I past_the_end) {
36     typedef typename std::iterator_traits<I>::value_type V;
37     I q;
38     for (q = first; q != past_the_end; ++q) {
39         *q = V(unsigned(std::rand()));
40     }
41 }
42
43 template <typename I>
44 void random_sequence(I first, I past_the_end, unsigned long seed) {
45     std::srand(seed);
46     typedef typename std::iterator_traits<I>::value_type V;

```

```

47  I q;
48  for (q = first; q != past_the_end; ++q) {
49      *q = V(unsigned(std::rand()));
50  }
51 }
52
53 int main() {
54     typedef Q::value_type E;
55
56     int const number_of_elements = NUMBER;
57     int repetitions = 100000 / number_of_elements;
58     if (repetitions < 3) {
59         repetitions = 3;
60     }
61
62     E* a = new E[number_of_elements];
63     DATA(&a[0], &a[number_of_elements]);
64
65     Q* many = new Q[repetitions];
66     std::clock_t start = std::clock();
67     for (int k = 0; k != repetitions; ++k) {
68         for (int i = 0; i != number_of_elements; ++i) {
69             (void) many[k].push(a[i]);
70         }
71     }
72     std::clock_t stop = std::clock();
73     for (int k = 0; k != repetitions; ++k) {
74         many[k].clear();
75     }
76     delete[] many;
77     delete[] a;
78
79     double running_time = double(stop - start)/double(CLOCKS_PER_SEC);
80     double time_per_run = 1000000.0 * running_time / double(repetitions);
81     double time_per_operation = time_per_run / double(number_of_elements);
82     std::cout << number_of_elements << " " << time_per_operation << "\n";
83
84     return 0;
85 }

```

§ 29 *push-comp.cpp*

```

1  #include <algorithm>
2  #include <cstdlib> // std::rand, std::srand
3  #include <ctime>
4
5  #ifndef NUMBER
6  #define NUMBER 1000000
7  #endif
8
9  #ifndef DATA
10 #define DATA increasing_sequence
11 #endif
12
13 #include "data-structure.i++" // Q comes from here
14 #include <iterator> // defines std::iterator_traits
15
16 template <typename I>
17 void increasing_sequence(I first, I past_the_end) {
18     typedef typename std::iterator_traits<I>::value_type E;
19     I q;
20     for (q = first; q != past_the_end; ++q) {
21         *q = E(unsigned(q - first));
22     }

```

```

23 }
24
25 template <typename I>
26 void decreasing_sequence(I first, I past_the_end) {
27     typedef typename std::iterator_traits<I>::value_type E;
28     I q;
29     for (q = first; q != past_the_end; ++q) {
30         *q = E(unsigned(past_the_end - q));
31     }
32 }
33
34 template <typename I>
35 void random_sequence(I first, I past_the_end) {
36     typedef typename std::iterator_traits<I>::value_type E;
37     I q;
38     for (q = first; q != past_the_end; ++q) {
39         *q = E(unsigned(std::rand()));
40     }
41 }
42
43 template <typename I>
44 void random_sequence(I first, I past_the_end, unsigned long seed) {
45     std::srand(seed);
46     typedef typename std::iterator_traits<I>::value_type E;
47     I q;
48     for (q = first; q != past_the_end; ++q) {
49         *q = E(unsigned(std::rand()));
50     }
51 }
52
53 int main() {
54     typedef Q::value_type E;
55
56     int const number_of_elements = NUMBER;
57
58     E* a = new E[number_of_elements];
59     DATA(&a[0], &a[number_of_elements]);
60
61     Q q;
62     for (int i = 0; i != number_of_elements; ++i) {
63         (void) q.push(a[i]);
64     }
65
66     double comp_count = double(comps);
67     double comp_per_operation = comp_count / double(number_of_elements);
68     std::cout << number_of_elements << " " << comp_per_operation << "\n";
69
70     q.clear();
71     delete[] a;
72     return 0;
73 }

```

§ 30 *increase-time.cpp*

```

1 #include <algorithm>
2 #include <cstdlib>
3 #include <ctime>
4 #include <vector>
5
6 #ifndef NUMBER
7 #define NUMBER 1000000
8 #endif
9
10 #include "data-structure.i++" // Q comes from here

```

```

11
12 int main() {
13     typedef Q::value_type E;
14
15     int const n = NUMBER;
16     int repetitions = 2000000 / n;
17     if (repetitions < 3) {
18         repetitions = 3;
19     }
20
21     E* a = new E[n];
22     for (int i = 0; i < n; i++) {
23         a[i] = (E) i;
24     }
25
26     srand(1);
27     for (int i = 0; i < n; i++) {
28         std::swap(a[i], a[rand() % (n)]);
29     }
30
31     std::vector<Q::iterator> v(n);
32     Q* many = new Q[repetitions];
33     std::clock_t start = std::clock();
34     for (int k = 0; k != repetitions; ++k) {
35         for (int i = 0; i != n; ++i) {
36             Q::iterator p = many[k].push(a[i]);
37             v[a[i]] = p;
38         }
39         Q::iterator p = many[k].top();
40     }
41     std::clock_t stop = std::clock();
42     double dual_time = double(stop - start)/double(CLOCKS_PER_SEC);
43     for (int k = 0; k != repetitions; ++k) {
44         many[k].clear();
45     }
46
47     start = std::clock();
48     for (int k = 0; k != repetitions; ++k) {
49         for (int i = 0; i != n; ++i) {
50             Q::iterator p = many[k].push(a[i]);
51             v[a[i]] = p;
52         }
53         Q::iterator p = many[k].top();
54         for (int i = 0; i != n; ++i) {
55             many[k].increase(v[i], i + n);
56         }
57     }
58     stop = std::clock();
59     for (int k = 0; k != repetitions; ++k) {
60         many[k].clear();
61     }
62     v.clear();
63     delete[] many;
64     delete[] a;
65
66     double running_time = double(stop - start)/double(CLOCKS_PER_SEC);
67     running_time -= dual_time;
68     double time_per_run = 1000000.0* running_time / double(repetitions);
69     double time_per_operation = time_per_run / double(n);
70     std::cout << n << " " << time_per_operation << "\n";
71
72     return 0;
73 }

```

§ 31 *increase-comp.cpp*

```

1 #include <algorithm>
2 #include <cmath>
3 #include <cstdlib>
4 #include <vector>
5
6 #ifndef NUMBER
7 #define NUMBER 1000000
8 #endif
9
10 #include "data-structure.i++" // Q comes from here
11
12 int main() {
13     typedef Q::value_type E;
14
15     int const n = NUMBER;
16     E* a = new E[n];
17     for (int i = 0; i < n; i++) {
18         a[i] = (E) i;
19     }
20
21     srand(1);
22     for (int i = 0; i < n; i++) {
23         std::swap(a[i], a[rand() % (n)]);
24     }
25
26     Q q;
27     std::vector<Q::iterator> v(n);
28     for (int i = 0; i != n; ++i) {
29         Q::iterator p = q.push(a[i]);
30         v[a[i]] = p;
31     }
32     Q::iterator p = q.top();
33
34     comps = 0;
35     for (int i = 0; i != n; ++i) {
36         q.increase(v[i], i + n);
37     }
38
39     double comp_count = double(comps);
40     double comp_per_operation = comp_count / double(n);
41     // double factor = comp_per_operation / std::log2(double(n));
42     std::cout << n << " " << comp_per_operation << "\n";
43
44     q.clear();
45     v.clear();
46     delete[] a;
47     return 0;
48 }

```

§ 32 *pop-time.cpp*

```

1 #include <algorithm>
2 #include <cstdlib>
3 #include <ctime>
4
5 #ifndef NUMBER
6 #define NUMBER 1000000
7 #endif
8
9 #include "data-structure.i++" // Q comes from here
10
11 int main() {

```

```

12 typedef Q::value_type E;
13
14 int const n = NUMBER;
15 int repetitions = 1000000 / n;
16 if (repetitions < 3) {
17     repetitions = 3;
18 }
19
20 Q q;
21 E* a = new E[n];
22
23 for (int i = 0; i < n; i++) {
24     a[i] = (E) i;
25 }
26
27 srand(1);
28 for (int i = 0; i < n; i++) {
29     std::swap(a[i], a[rand() % (n)]);
30 }
31
32 Q* many = new Q[repetitions];
33 std::clock_t start = std::clock();
34 for (int k = 0; k != repetitions; ++k) {
35     for (int i = 0; i != n; ++i) {
36         (void) many[k].push(a[i]);
37     }
38 }
39 std::clock_t stop = std::clock();
40 double dual_time = double(stop - start)/double(CLOCKS_PER_SEC);
41 for (int k = 0; k != repetitions; ++k) {
42     many[k].clear();
43 }
44
45 start = std::clock();
46 for (int k = 0; k != repetitions; ++k) {
47     for (int i = 0; i != n; ++i) {
48         (void) many[k].push(a[i]);
49     }
50     for (int i = 0; i != n; ++i) {
51         many[k].pop();
52     }
53 }
54 stop = std::clock();
55 delete[] many;
56 delete[] a;
57
58 double running_time = double(stop - start)/double(CLOCKS_PER_SEC);
59 running_time -= dual_time;
60 double time_per_run = 1000000.0 * running_time / double(repetitions);
61 double time_per_operation = time_per_run / double(n);
62 std::cout << n << " " << time_per_operation << "\n";
63
64 return 0;
65 }

```

§ 33 *pop-comp.cpp*

```

1 #include <algorithm>
2 #include <cmath>
3 #include <cstdlib>
4
5 #ifndef NUMBER
6 #define NUMBER 1000000
7 #endif

```

```

8
9 #include "data-structure.i++" // Q comes from here
10
11 int main() {
12     typedef Q::value_type E;
13
14     int const n = NUMBER;
15     E* a = new E[n];
16     for (int i = 0; i < n; i++) {
17         a[i] = (E) i;
18     }
19     srand(1);
20     for (int i = 0; i < n; i++) {
21         std::swap(a[i], a[rand() % (n)]);
22     }
23
24     Q q;
25     for (int i = 0; i != n; ++i) {
26         (void) q.push(a[i]);
27     }
28
29     comps = 0;
30     for (int i = 0; i != n; ++i) {
31         q.pop();
32     }
33
34     double comp_count = double(comps);
35     double comp_per_operation = comp_count / double(n);
36     double comp_per_n_lg_n = comp_per_operation / std::log2(double(n));
37     std::cout << n << " " << comp_per_operation << "\n";
38
39     q.clear();
40     delete[] a;
41     return 0;
42 }

```

Makefile

§ 34 *benchmark.mk*

```

1 CPHSTL_ROOT=$(HOME)/CPHSTL
2 CXXFLAGS = -DDEBUG -D_GNU__ -Wall -std=c++0x -pedantic -x c++ -fno-strict-
   aliasing -O3
3 DEBUGFLAGS = -DDEBUG -D_GNU__ -Wall -std=c++0x -pedantic -x c++ -g
4 IFLAGS = -I $(CPHSTL_ROOT)/Source/Meldable-priority-queue/Code -I $(CPHSTL_ROOT)/
   Source/Assert/Code -I $(CPHSTL_ROOT)/Source/Meldable-priority-queue/Code -I
   ../Code -I $(CPHSTL_ROOT)/Source/Priority-queue-frameworks/Code -I $(
   CPHSTL_ROOT)/Source/Priority-queue-frameworks/Adaptive-sorting -I $(
   CPHSTL_ROOT)/Source/Iterator/Code -I $(CPHSTL_ROOT)/Source/Proxy/Code -I $(
   CPHSTL_ROOT)/Source/Type/Code -I $(CPHSTL_ROOT)/Progress/Meldable-priority-
   queue/Stefan-fibonacci-heap -I . -I $(LEDAROOT)/incl -I$(LEDAROOT)
5 LFLAGS = -lleda -lX11 -lm
6 CXX = g++
7
8 default:
9
10 implementation-files:= $(wildcard *.i++)
11 data-structures:= $(basename $(implementation-files))
12 time-tests = $(addsuffix .time, $(data-structures))
13 comp-tests = $(addsuffix .comp, $(data-structures))
14 dijkstra-tests:= $(addsuffix .dijkstra, $(data-structures))
15 push-time-tests:= $(addsuffix .push, $(time-tests))
16 push-comp-tests:= $(addsuffix .push, $(comp-tests))
17 increase-time-tests:= $(addsuffix .increase, $(time-tests))

```

```

18 increase-comp-tests:= $(addsuffix .increase, $(comp-tests))
19 pop-time-tests:= $(addsuffix .pop, $(time-tests))
20 pop-comp-tests:= $(addsuffix .pop, $(comp-tests))
21
22 $(time-tests): %.time: %.i++
23     @make -s $*.time.push
24     @make -s $*.time.increase
25     @make -s $*.time.pop
26
27 $(comp-tests): %.comp: %.i++
28     @make -s $*.comp.push
29     @make -s $*.comp.increase
30     @make -s $*.comp.pop
31
32 list = 10000 100000 1000000
33 data = "increasing_sequence"
34 # "decreasing_sequence" "random_sequence"
35
36 $(dijkstra-tests): %.dijkstra : %.i++
37     @echo $*: time and comps for dijkstra"
38     @cp $*.h++ data-structure.h++
39     @for x in $(list) ; do \
40         $(CXX) $(CXXFLAGS) -DNODES=$$x $(IFLAGS) dijkstra.c++ $(LFLAGS);\
41         ./a.out; \
42         rm -f ./a.out ; \
43     done
44
45 $(push-time-tests): %.time.push : %.i++
46     @echo $* "time per push"
47     @cp $*.i++ data-structure.i++
48     @for d in $(data) ; do \
49         echo $$d; \
50         for x in $(list) ; do \
51             $(CXX) $(CXXFLAGS) -DNUMBER=$$x -DDATA=$$d $(IFLAGS) push-time.c++ $
52                 (LFLAGS); \
53                 ./a.out; \
54                 rm -f ./a.out ; \
55         done \
56     done
57
58 $(increase-time-tests): %.time.increase : %.i++
59     @echo $* "time per increase"
60     @cp $*.i++ data-structure.i++
61     @for x in $(list) ; do \
62         $(CXX) $(CXXFLAGS) -DNUMBER=$$x $(IFLAGS) increase-time.c++ $(LFLAGS)
63         ; \
64         ./a.out; \
65         rm -f ./a.out ; \
66     done
67     @rm data-structure.i++
68
69 $(pop-time-tests): %.time.pop : %.i++
70     @echo $* "time per pop"
71     @cp $*.i++ data-structure.i++
72     @for x in $(list) ; do \
73         $(CXX) $(CXXFLAGS) -DNUMBER=$$x $(IFLAGS) pop-time.c++ $(LFLAGS);\
74         ./a.out; \
75         rm -f ./a.out ; \
76     done
77     @rm data-structure.i++
78
79 $(push-comp-tests): %.comp.push : %.i++
80     @echo $* "# comp per push"
81     @cp $*.i++ data-structure.i++
82     @for d in $(data) ; do \

```

```

81     for x in $(list) ; do \
82         $(CXX) $(CXXFLAGS) -DNUMBER=$$x -DDATA=$$d $(IFLAGS) push-comp.c++ $
            (LFLAGS);\
83         ./a.out; \
84         rm -f ./a.out ; \
85     done \
86 done
87 @rm data-structure.i++
88
89 $(increase-comp-tests): %.comp.increase : %.i++
90     @echo $* "# comp per increase"
91     @cp $*.i++ data-structure.i++
92     @for x in $(list) ; do \
93         $(CXX) $(CXXFLAGS) -DNUMBER=$$x $(IFLAGS) increase-comp.c++ $(LFLAGS)
            ;\
94         ./a.out; \
95         rm -f ./a.out ; \
96     done
97     @rm data-structure.i++
98
99 $(pop-comp-tests): %.comp.pop : %.i++
100     @echo $* "# comp per pop"
101     @cp $*.i++ data-structure.i++
102     @for x in $(list) ; do \
103         $(CXX) $(CXXFLAGS) -DNUMBER=$$x $(IFLAGS) pop-comp.c++ $(LFLAGS);\
104         ./a.out; \
105         rm -f ./a.out ; \
106     done
107     @rm data-structure.i++
108
109 clean:
110     @rm -vf *~ a.out temp plot.*

```