Priority-queue frameworks: Programs

Jyrki Katajainen

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

Abstract. This is an electronic appendix to the article "Policy-based benchmarking of weak heaps and their relatives". The report contains the programs related to our priority-queue frameworks. Look at the CPH STL reports 2009-3 and 2009-4 to see examples of other component frameworks.

Keywords. CPH STL, component frameworks, priority queues, weak heaps, weak queues, run-relaxed weak queues, rank-relaxed weak queues

CPH STL Report 2009-7, October 2009. Revised January 2010, February 2010, March 2010.

Table of contents

1 Lines-of-code (LOC) counts for the files 1
2 Benchmark results 1
3 Meldable priority queue 3
3.1 Meldable-priority-queue/Code/stl-meldable-priority-queue.h++ 3
3.2 Meldable-priority-queue/Code/stl-meldable-priority-queue.i++ 6
4 Priority-queue frameworks 10
4.1 Priority-queue-frameworks/Code/single-heap-framework.h++ 10
4.2 Priority-queue-frameworks/Code/multiple-heap-framework.h++ 14
4.3 Priority-queue-frameworks/Code/multiple-heap-framework.i++ 16
5 Heapifiers 24
5.1 Priority-queue-frameworks/Code/top-down-binary-heap-heapifier.h++ 24
5.2 Priority-queue-frameworks/Code/weak-heap-heapifier.h++ . . 25
6 Encapsulators 27
6.1 Priority-queue-frameworks/Code/element-encapsulator.h++ . 27
6.2 Priority-queue-frameworks/Code/heap-node.h++ 27
6.3 Priority-queue-frameworks/Code/weak-heap-node.h++ 35
6.4 Priority-queue-frameworks/Code/fat-weak-heap-node.h++ . . 36
7 Heap store 42
7.1 Priority-queue-frameworks/Code/heap-proxy.h++ 42
7.2 Priority-queue-frameworks/Code/proxy-list-heap-store.h++ . 44
8 Mark stores 50
8.1 Priority-queue-frameworks/Code/blank-mark-store.h++ 50
8.2 Priority-queue-frameworks/Code/lazy-mark-store.h++ 52
8.3 Priority-queue-frameworks/Code/eager-mark-store.h++ 70
9 Bit hacks 76
9.1 Priority-queue-frameworks/Code/bit-manipulation.h++ 76
9.2 Priority-queue-frameworks/Code/bit-store.h++ 78
10 Iterator 81
10.1 Iterator/Code/priority-queue-iterator.h++ 81
10.2 Iterator/Code/priority-queue-iterator.i++ 83
10.3 Iterator/Code/proxy-iterator.h++ 86
10.4 Iterator/Code/proxy-iterator.i++ 88
11 Proxies 92
11.1 Proxy/Code/allocator-proxy.h++ 92

11.2 Proxy/Code/comparator-proxy.h++ 94
11.3 Proxy/Code/vector-surrogate.h++ 94
12 Unit tests 95
12.1 Assert/Code/assert.h++ 95
12.2 Meldable-priority-queue/Test/smoke-test.c++ 96
12.3 Meldable-priority-queue/Test/test-case.c++ 100
12.4 Meldable-priority-queue/Test/unittest.mk 102
13 Benchmarks 103
13.1 Priority-queue-frameworks/Benchmark/push-comp.c++ 103
13.2 Priority-queue-frameworks/Benchmark/increase-comp.c++ . . 103
13.3 Priority-queue-frameworks/Benchmark/erase-comp.c++ 104
13.4 Priority-queue-frameworks/Benchmark/pop-comp.c++ 105
13.5 Priority-queue-frameworks/Benchmark/push-time.c++ 105
13.6 Priority-queue-frameworks/Benchmark/increase-time.c++ . . . 106
13.7 Priority-queue-frameworks/Benchmark/erase-time.c++ 107
13.8 Priority-queue-frameworks/Benchmark/pop-time.c++ 108
13.9 Priority-queue-frameworks/Benchmark/first-version.i++ 109
13.10 Priority-queue-frameworks/Benchmark/binary-heap.i++ 110
13.11 Priority-queue-frameworks/Benchmark/weak-heap.i++ 110
13.12 Priority-queue-frameworks/Benchmark/weak-queue.i++ 111
13.13 Priority-queue-frameworks/Benchmark/run-relaxed-weak-queue.i++ 111
13.14 Priority-queue-frameworks/Benchmark/rank-relaxed-weak-queue.i++ 111
13.15 Priority-queue-frameworks/Benchmark/fibonacci-heap.i++ . . 112
13.16 Priority-queue-frameworks/Benchmark/pairing-heap.i++ . . . 113
13.17 Priority-queue-frameworks/Benchmark/benchmark.mk 114
13.18 Priority-queue-frameworks/Benchmark/oscilation.py 116

1. Lines-of-code (LOC) counts for the files

In our use of the LOC metric we ignore comment lines and lines with a single parenthesis, and we calculate long statements as single lines. Also, all old versions, debugging code, and assertions are excluded from these counts.

Table 1. LOC counts for our files.

File	LOC
`stl-meldable-priority-queue.h++`	67
`stl-meldable-priority-queue.i++`	156
`single-heap-framework.h++`	100
`multiple-heap-framework.h++`	49
`multiple-heap-framework.i++`	135
`top-down-binary-heap-heapifier.h++`	36
`weak-heap-heapifier.h++`	55
`element-encapsulator.h++`	23
`heap-node.h++`	243
`weak-heap-node.h++`	19
`fat-weak-heap-node.h++`	72
`heap-proxy.h++`	44
`proxy-list-heap-store.h++`	137
`blank-mark-store.h++`	56
`lazy-mark-store.h++`	525
`eager-mark-store.h++`	179
`bit-manipulation.h++`	30
`bit-store.h++`	84
`priority-queue-iterator.h++`	61
`priority-queue-iterator.i++`	77
`proxy-iterator.h++`	70
`proxy-iterator.i++`	143
`allocator-proxy.h++`	53
`comparator-proxy.h++`	29
`vector-surrogate.h++`	16
`binary-heap.i++`	22
`weak-heap.i++`	24
`weak-queue.i++`	11
`run-relaxed-weak-queue.i++`	24
`rank-relaxed-weak-queue.i++`	24
`fibonacci-heap.i++`	46
`pairing-heap.i++`	46

2. Benchmark results

In the preliminary experiments I compared the tuned versions of weak heap, weak queue, run-relaxed weak queue, and rank-relaxed weak queue. The input data was of type `long long`. (For more details about the benchmarks,

see Section 13.) The results obtained are listed in a raw form below. Running times are given in microseconds. The experiments were performed on my laptop computer (model Intel® Core™2 CPU T5600 @ 1.83GHz) under Ubuntu 9.10 (Linux kernel 2.6.31-18-generic) using `g++` (`gcc` version 4.4.1 with options `-DNDEBUG -Wall -std=c++0x -pedantic -x c++ -O3`). The size of L2 cache of this computer is about 2 MB and that of the main memory 1 GB.

```
weak-heap time per push
10000 0.19
100000 0.21
1000000 0.26
weak-heap time per increase
10000 0.29
100000 0.62
1000000 1.44
weak-heap time per erase
10000 0.19
100000 0.16
1000000 0.35
weak-heap time per pop
10000 0.4
100000 0.67
1000000 1.84

weak-queue time per push
10000 0.11
100000 0.11
1000000 0.114
weak-queue time per increase
10000 0.27
100000 0.5
1000000 1.574
weak-queue time per erase
10000 0.3
100000 0.38
1000000 0.372
weak-queue time per pop
10000 0.46
100000 0.73
1000000 1.792

run-relaxed-weak-queue time per push
10000 0.13
100000 0.14
1000000 0.124
run-relaxed-weak-queue time per increase
```

```
10000 0.66
100000 0.84
1000000 1.244
run-relaxed-weak-queue time per erase
10000 0.83
100000 0.88
1000000 0.936
run-relaxed-weak-queue time per pop
10000 0.56
100000 1.08
1000000 2.402

rank-relaxed-weak-queue time per push
10000 0.13
100000 0.12
1000000 0.136
rank-relaxed-weak-queue time per increase
10000 0.38
100000 0.65
1000000 0.98
rank-relaxed-weak-queue time per erase
10000 0.7
100000 0.68
1000000 0.726
rank-relaxed-weak-queue time per pop
10000 0.56
100000 1.07
1000000 2.43
```

3. Meldable priority queue

3.1 Meldable-priority-queue/Code/stl-meldable-priority-queue.h++

```
1  /*
2    A meldable priority queue is a container which provides forward
3    iterators to the elements stored.
4
5    CPH STL guarantees:
6
7    1) Member function push() returns an iterator (or a handle) to the
8    given element and this iterator remains valid the whole life time of
9    the element.
10
11   2) Iterator operations take logarithmic time in the worst case.
12
13   3) Member function top() is a constant-time operation.
14
15   4) Member functions push(), pop(), erase(), and increase() have the
16   logarithmic worst-case cost.
17
18   5) Member function meld() is relatively fast having a
```

```
19    polylogarithmic worst-case cost.
20
21    6) The data structure uses a linear number of words in addition to
22    the elements stored.
23
24    Container requirements not fulfilled [C++ standard §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  #include <functional> // std::less
39  #include <memory> // std::allocator
40  #include "multiple-heap-framework.h++"
41  #include "priority-queue-iterator.h++"
42  #include "weak-heap-node.h++"
43
44  namespace cphstl {
45
46    template <
47      typename V,
48      typename C = std::less<V>,
49      typename A = std::allocator<V>,
50      typename E = cphstl::weak_heap_node<V, A>,
51      typename R = cphstl::multiple_heap_framework<V, C, A, E>,
52      typename I = cphstl::priority_queue_iterator<E, R>,
53      typename J = cphstl::priority_queue_iterator<E, R>
54    >
55    class meldable_priority_queue {
56    public:
57
58      // types
59
60      typedef V value_type;
61      typedef C comparator_type;
62      typedef A allocator_type;
63      typedef E encapsulator_type;
64      typedef R realizator_type;
65      typedef I iterator;
66      typedef J const_iterator;
67      typedef typename R::reference reference;
68      typedef typename R::const_reference const_reference;
69      typedef V* pointer;
70      typedef V const* const_pointer;
71      typedef std::size_t size_type;
72      typedef std::ptrdiff_t difference_type;
73      typedef std::reverse_iterator<iterator> reverse_iterator;
74      typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
75      typedef meldable_priority_queue<V, C, A, E, R, I, J> container_type;
76
77    protected:
78
79      typedef typename A::template rebind<E>::other encapsulator_allocator_type;
80
81      encapsulator_allocator_type allocator;
82      R realizator;
83
```

```
84      E* create(V const&);
85      void destroy(E*);
86
87      template <typename K>
88      void insert(K, K);
89
90    public:
91
92      // structors
93
94      explicit meldable_priority_queue(C const& = C(), A const& = A());
95      meldable_priority_queue(meldable_priority_queue const&);
96      meldable_priority_queue& operator=(meldable_priority_queue const&);
97      ~meldable_priority_queue();
98
99      // iterators
100
101     iterator begin();
102     const_iterator begin() const;
103     iterator end();
104     const_iterator end() const;
105
106     // accessors
107
108     allocator_type get_allocator() const;
109     comparator_type get_comparator() const;
110     bool empty() const;
111     size_type size() const;
112     size_type max_size() const;
113     const_iterator top() const;
114
115     // modifiers
116
117     iterator top();
118     iterator push(V const&);
119     void pop();
120     void erase(iterator);
121     void increase(iterator, V const&);
122     void clear();
123     void meld(meldable_priority_queue&);
124     void swap(meldable_priority_queue&);
125
126 #ifdef DEBUG
127
128     bool is_valid() {
129       return realizator.is_valid();
130     }
131
132     void show() {
133       realizator.show();
134     }
135
136 #endif
137
138   };
139
140   // algorithms
141
142   template <typename V, typename C, typename A, typename E,
143             typename R, typename I, typename J>
144   meldable_priority_queue<V, C, A, E, R, I, J>&
145   meld(meldable_priority_queue<V, C, A, E, R, I, J>&,
146        meldable_priority_queue<V, C, A, E, R, I, J>&);
147
148   template <typename V, typename C, typename A, typename E,
```

```
149             typename R, typename I, typename J>
150   void swap(meldable_priority_queue<V, C, A, E, R, I, J>&,
151             meldable_priority_queue<V, C, A, E, R, I, J>&);
152 }
153
154 #include "stl-meldable-priority-queue.i++"
155 #endif
```

3.2 Meldable-priority-queue/Code/stl-meldable-priority-queue.i++

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

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

```
121   }
122
123   template <typename V, typename C, typename A, typename E,
124             typename R, typename I, typename J>
125   typename meldable_priority_queue<V, C, A, E, R, I, J>::iterator
126   meldable_priority_queue<V, C, A, E, R, I, J>::begin() {
127     return iterator(realizator.begin(), &realizator);
128   }
129
130   template <typename V, typename C, typename A, typename E,
131             typename R, typename I, typename J>
132   typename meldable_priority_queue<V, C, A, E, R, I, J>::const_iterator
133   meldable_priority_queue<V, C, A, E, R, I, J>::begin() const {
134     return const_iterator(realizator.begin(), (R*) &realizator);
135   }
136
137   template <typename V, typename C, typename A, typename E,
138             typename R, typename I, typename J>
139   typename meldable_priority_queue<V, C, A, E, R, I, J>::iterator
140   meldable_priority_queue<V, C, A, E, R, I, J>::end() {
141     return iterator(realizator.end(), &realizator);
142   }
143
144   template <typename V, typename C, typename A, typename E,
145             typename R, typename I, typename J>
146   typename meldable_priority_queue<V, C, A, E, R, I, J>::const_iterator
147   meldable_priority_queue<V, C, A, E, R, I, J>::end() const {
148     return const_iterator(realizator.end(), (R*) &realizator);
149   }
150
151   template <typename V, typename C, typename A, typename E,
152             typename R, typename I, typename J>
153   bool
154   meldable_priority_queue<V, C, A, E, R, I, J>::empty() const {
155     return (*this).size() == size_type(0);
156   }
157
158   template <typename V, typename C, typename A, typename E,
159             typename R, typename I, typename J>
160   typename meldable_priority_queue<V, C, A, E, R, I, J>::size_type
161   meldable_priority_queue<V, C, A, E, R, I, J>::size() const {
162     return realizator.size();
163   }
164
165   template <typename V, typename C, typename A, typename E,
166             typename R, typename I, typename J>
167   typename meldable_priority_queue<V, C, A, E, R, I, J>::size_type
168   meldable_priority_queue<V, C, A, E, R, I, J>::max_size() const {
169     return realizator.max_size();
170   }
171
172   template <typename V, typename C, typename A, typename E,
173             typename R, typename I, typename J>
174   typename meldable_priority_queue<V, C, A, E, R, I, J>::const_iterator
175   meldable_priority_queue<V, C, A, E, R, I, J>::top() const {
176     return const_iterator(realizator.top(), &realizator);
177   }
178
179   template <typename V, typename C, typename A, typename E,
180             typename R, typename I, typename J>
181   typename meldable_priority_queue<V, C, A, E, R, I, J>::iterator
182   meldable_priority_queue<V, C, A, E, R, I, J>::top() {
183     return iterator(realizator.top(), &realizator);
184   }
185
```

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

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

4. Priority-queue frameworks

4.1 Priority-queue-frameworks/Code/single-heap-framework.h++

```
 1 /*
 2   A priority-queue framework for a single array-based heap.
 3
 4   Authors: Stefan Edelkamp, Jyrki Katajainen © 2010
 5 */
 6
 7 #ifndef __CPHSTL_SINGLE_HEAP_FRAMEWORK__
 8 #define __CPHSTL_SINGLE_HEAP_FRAMEWORK__
 9
10 #include <algorithm>
11 #include "allocator-proxy.h++"
12 #include "assert.h++"
13 #include "comparator-proxy.h++"
14 #include <cstddef>
15 #include <cstdlib>
16 #include "element-encapsulator.h++"
17 #include <functional>
18 #include <iostream>
19 #include <memory>
20 #include "top-down-binary-heap-heapifier.h++"
21 #include <vector>
22 #include "vector-surrogate.h++"
23 #include <utility>
24
25 namespace cphstl {
26
27   template <
28     typename V,
29     typename C = std::less<V>,
30     typename A = std::allocator<V>,
31     typename E = element_encapsulator<V, std::ptrdiff_t, A>,
32     typename H = top_down_binary_heap_heapifier,
33     typename K = std::vector<E*, A>,
34     typename S = vector_surrogate<K>
35   >
36   class single_heap_framework {
37   public:
38
39     // types
40
41     typedef V value_type;
42     typedef C comparator_type;
43     typedef A allocator_type;
44     typedef E encapsulator_type;
45     typedef H heapifier_type;
46     typedef K kernel_type;
47     typedef S surrogate_type;
48     typedef std::size_t size_type;
```

```
49    typedef std::ptrdiff_t difference_type;
50    typedef V& reference;
51    typedef V const& const_reference;
52
53  protected:
54
55    typedef typename A::template rebind<S>::other surrogate_allocator_type;
56
57    comparator_proxy<C> comparator;
58    allocator_proxy<surrogate_allocator_type> surrogate_allocator;
59    K vector;
60    H heapifier;
61
62  public:
63
64    S* surrogate;
65
66    // structors
67
68    explicit single_heap_framework(C const& c = C(), A const& a = A(),
69                                   H const& h = H())
70      : comparator(c), surrogate_allocator(a), vector(a), heapifier(h) {
71      surrogate = surrogate_allocator.allocate(1);
72      (*surrogate).subject() = &vector;
73    }
74
75    ~single_heap_framework() {
76      surrogate_allocator.deallocate(surrogate, 1);
77    }
78
79    // iterators
80
81    E* begin() const {
82      if (vector.size() == 0) {
83        return (E*) 0;
84      }
85      return vector[0];
86    }
87
88    E* end() const {
89      return (E*) 0;
90    }
91
92    // accessors
93
94    A get_allocator() const {
95      return A(vector.get_allocator());
96    }
97
98    C get_comparator() const {
99      return comparator.subject();
100   }
101
102   size_type size() const {
103     return vector.size();
104   }
105
106   size_type max_size() const {
107     typename std::vector<int, A>::allocator_type a;
108     size_type available_memory = a.max_size() * sizeof(int); // in bytes
109     return available_memory / (sizeof(E) + 2 * sizeof(E*));
110   }
111
112   E* top() const {
113     return vector[0];
```

```
114     }
115
116     // modifiers
117
118     void insert(E* node) {
119       size_type last = vector.size();
120       (*node).position() = last;
121       vector.push_back(node);
122       heapifier.siftup(comparator, vector, last);
123     }
124
125     E* extract() {
126       assert(vector.size() != 0);
127       E* node = vector.back();
128       vector.pop_back();
129       return node;
130     }
131
132     void extract(E* node) {
133       if (vector.size() == 1) {
134         assert(vector[0] == node);
135         vector.pop_back();
136         return;
137       }
138       size_type i = std::abs(difference_type((*node).position()));
139       size_type last = vector.size() - 1;
140       std::swap(vector[i], vector[last]);
141       std::swap(vector[i]->position(), vector[last]->position());
142       size_type j = heapifier.siftdown(comparator, vector, last, i);
143       if (i == j ) {
144         heapifier.siftup(comparator, vector, j);
145       }
146       vector.pop_back();
147     }
148
149     void increase(E* node, V const& v) {
150       assert(! comparator(v, (*node).element()));
151       (*node).element() = v;
152       size_type i = std::abs(difference_type((*node).position()));
153       heapifier.siftup(comparator, vector, i);
154     }
155
156     void meld(single_heap_framework& other) {
157       if (size() > other.size()) {
158         swap(other);
159       }
160       while (other.vector.size() != 0) {
161         insert(other.extract());
162       }
163     }
164
165     void swap(single_heap_framework& other) {
166       std::swap(comparator, other.comparator);
167       std::swap(surrogate_allocator, other.surrogate_allocator);
168       std::swap(vector, other.vector);
169       std::swap(heapifier, other.heapifier);
170       std::swap(surrogate, other.surrogate);
171     }
172
173 #ifdef DEBUG
174
175     size_type parent(size_type j) {
176       return ((j - 1) / 2);
177     }
178
```

```
179     bool is_valid() {
180       return heapifier.is_valid(comparator, vector);
181     }
182 
183     void show() {
184       size_type size = vector.size();
185       std::cout << std::endl << "size: " << size << std::endl;
186       std::cout << "values:" << std::endl;
187       for (size_type i = 0; i < size; ++i) {
188         std::cout << vector[i]->element() << " ";
189       }
190       std::cout << "\n";
191       std::cout << "indices:" << std::endl;
192       for (size_type i = 0; i < size; ++i) {
193         std::cout << vector[i]->position() << " ";
194       }
195       std::cout << "\n";
196       std::cout << "\n ------------------------------------ \n";
197     }
198 
199 #endif
200 
201   };
202 }
203 
204 #endif
205 #ifdef UNITTEST_SINGLE_HEAP_FRAMEWORK
206 
207 #include "weak-heap-heapifier.h++"
208 
209 long comps = 0;
210 
211 template <typename V>
212 class counting_comparator {
213 public:
214 
215   bool operator()(V const& a, V const& b) const {
216     ++comps;
217     return a < b;
218   }
219 };
220 
221 template <typename R>
222 void unittest_single_heap_framework() {
223   typedef typename R::encapsulator_type E;
224   typedef typename E::value_type V;
225   typedef std::size_t size_type;
226 
227   R heap;
228 
229   std::vector<E*> v;
230   size_type number_of_elements = 1000000;
231   comps = 0;
232   size_type total = 0;
233   std::cout<< "n: " << number_of_elements << std::endl;
234 
235   for (size_type i = 0; i < number_of_elements; i++) {
236     E* p = new E(V(rand() % 100));
237     heap.insert(p);
238     v.push_back(p);
239     assert(heap.is_valid());
240   }
241   std::cout << "insert: comps per operation: "
242             << (double) comps / number_of_elements << std::endl;
243   total += comps;
```

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

4.2 Priority-queue-frameworks/Code/multiple-heap-framework.h++

```
 1 /*
 2   A priority-queue framework can be used to generate a realizator for
 3   a meldable priority queue. The efficiency of different operations
 4   depends on the policies relied on.
 5
 6   Author: Jyrki Katajainen © 2009, 2010
 7 */
 8
 9 #ifndef __CPHSTL_MULTIPLE_HEAP_FRAMEWORK__
10 #define __CPHSTL_MULTIPLE_HEAP_FRAMEWORK__
11
12 #include "allocator-proxy.h++"
13 #include "blank-mark-store.h++"
14 #include "comparator-proxy.h++"
15 #include <cstddef> // std::size_t and std::ptrdiff_t
16 #include "proxy-list-heap-store.h++"
17 #include <functional> // std::less
```

```
18 #include <memory> // std::allocator
19 #include "weak-heap-node.h++"
20
21 namespace cphstl {
22
23   template <
24     typename V,
25     typename C = std::less<V>,
26     typename A = std::allocator<V>,
27     typename E = weak_heap_node<V, A>,
28     typename H = proxy_list_heap_store<C, A, E>,
29     typename M = blank_mark_store<C, A, E>
30   >
31   class multiple_heap_framework {
32   public:
33
34     // types
35
36     typedef V value_type;
37     typedef C comparator_type;
38     typedef A allocator_type;
39     typedef E encapsulator_type;
40     typedef H heap_store_type;
41     typedef M mark_store_type;
42     typedef std::size_t size_type;
43     typedef V& reference;
44     typedef V const& const_reference;
45
46     // structors
47
48     explicit multiple_heap_framework(C const& = C(), A const& = A());
49     ~multiple_heap_framework();
50
51     // iterators
52
53     E* begin() const;
54     E* end() const;
55
56     // accessors
57
58     A get_allocator() const;
59     C get_comparator() const;
60     size_type size() const;
61     size_type max_size() const;
62     E* top() const;
63
64     // modifiers
65
66     E* insert(E*);
67     E* extract();
68     void extract(E*);
69     void increase(E*, V const&);
70     void meld(multiple_heap_framework&);
71     void swap(multiple_heap_framework&);
72
73 #ifdef DEBUG
74
75     void show() {
76       heap_store.show();
77     }
78
79 #endif
80
81   protected:
82
```

```
83      comparator_proxy<C> comparator;
84      allocator_proxy<A> allocator;
85      H heap_store;
86      M mark_store;
87      E* top_;
88      size_type size_;
89
90    private:
91
92      multiple_heap_framework(multiple_heap_framework const&);
93      multiple_heap_framework& operator=(multiple_heap_framework const&);
94    };
95
96  }
97
98  #include "multiple-heap-framework.i++"
99  #endif
```

4.3 Priority-queue-frameworks/Code/multiple-heap-framework.i++

```
1   /*
2     Implementation of the multiple-heap priority-queue framework.
3
4     Author: Jyrki Katajainen © 2009, 2010
5   */
6
7   #include <algorithm> // std::swap
8   #include "assert.h++"
9   #include <climits> // LONG_MAX
10  #include <iostream>
11  #include <list>
12  #include <vector>
13  #include <utility> // std::pair
14
15  namespace cphstl {
16
17    template <typename V, typename C, typename A, typename E,
18              typename H, typename M>
19    multiple_heap_framework<V, C, A, E, H, M>::multiple_heap_framework(
20      C const& c, A const& a)
21      : comparator(c), allocator(a), heap_store(c, a), mark_store(c), top_(0),
22        size_(0) {
23    }
24
25    template <typename V, typename C, typename A, typename E,
26              typename H, typename M>
27    multiple_heap_framework<V, C, A, E, H, M>::~multiple_heap_framework() {
28      // Precondition: The data structure contains no elements.
29    }
30
31    template <typename V, typename C, typename A, typename E,
32              typename H, typename M>
33    E*
34    multiple_heap_framework<V, C, A, E, H, M>::begin() const {
35      if (size_ == 0) {
36        return (E*) 0;
37      }
38      return (*heap_store.begin()).root();
39    }
40
41    template <typename V, typename C, typename A, typename E,
42              typename H, typename M>
43    E*
44    multiple_heap_framework<V, C, A, E, H, M>::end() const {
45      return (E*) 0;
```

```
46   }
47
48   template <typename V, typename C, typename A, typename E,
49             typename H, typename M>
50   A
51   multiple_heap_framework<V, C, A, E, H, M>::get_allocator() const {
52     return allocator.subject();
53   }
54
55   template <typename V, typename C, typename A, typename E,
56             typename H, typename M>
57   C
58   multiple_heap_framework<V, C, A, E, H, M>::get_comparator() const {
59     return comparator.subject();
60   }
61
62   template <typename V, typename C, typename A, typename E,
63             typename H, typename M>
64   typename multiple_heap_framework<V, C, A, E, H, M>::size_type
65   multiple_heap_framework<V, C, A, E, H, M>::size() const {
66     return size_;
67   }
68
69   template <typename V, typename C, typename A, typename E,
70             typename H, typename M>
71   typename multiple_heap_framework<V, C, A, E, H, M>::size_type
72   multiple_heap_framework<V, C, A, E, H, M>::max_size() const {
73     // Assume that n <= LONG_MAX.
74     typename std::vector<int, A>::allocator_type a;
75     size_type n = LONG_MAX;
76     size_type available_memory = a.max_size() * sizeof(int); // in bytes
77     size_type space_available_for_encapsulators = available_memory -
78       heap_store.footprint(n) - mark_store.footprint(n);
79     return space_available_for_encapsulators / E::footprint();
80   }
81
82   template <typename V, typename C, typename A, typename E,
83             typename H, typename M>
84   E*
85   multiple_heap_framework<V, C, A, E, H, M>::top() const {
86     return top_;
87   }
88
89   template <typename V, typename C, typename A, typename E,
90             typename H, typename M>
91   E*
92   multiple_heap_framework<V, C, A, E, H, M>::insert(E* p) {
93
94     heap_store.inject(p, 0, mark_store);
95     if (top_ == 0 || comparator((*top_).element(), (*p).element())) {
96       top_ = p;
97     }
98     ++size_;
99
100 #ifdef DEBUG
101
102     assert(heap_store.is_valid(mark_store));
103     assert(mark_store.is_valid());
104
105 #endif
106
107     return p;
108  }
109
110  template <typename V, typename C, typename A, typename E,
```

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

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

```
241       E* s = q;
242       q = (*q).parent();
243       r = (*r).fast_join(s, replacement, comparator, mark_store);
244     }
245     (*r).splice_in(hole_at_p, heap_store);
246     mark_store.mark(r);
247     mark_store.reduce(heap_store);
248     if (p == top_) {
249       top_ = heap_store.find_top();
250       E* t = mark_store.find_top();
251       if (t != 0 && comparator((*top_).element(), (*t).element())) {
252         top_ = t;
253       }
254     }
255 
256 #ifdef DEBUG
257 
258     assert(heap_store.is_valid(mark_store));
259     assert(mark_store.is_valid());
260 
261 #endif
262 
263   }
264 
265 #endif
266 
267   template <typename V, typename C, typename A, typename E,
268             typename H, typename M>
269   void
270   multiple_heap_framework<V, C, A, E, H, M>::meld(
271     multiple_heap_framework& other) {
272     // Precondition: The allocators and comparators must be compatible.
273     if (size_ < other.size_) {
274       swap(other);
275     }
276     heap_store.meld(other.heap_store, mark_store);
277     mark_store.meld(other.mark_store, heap_store);
278     size_ += other.size_;
279     other.size_ = 0;
280     if (other.top_ != 0 &&
281         comparator((*top_).element(), (*other.top_).element())) {
282       top_ = other.top_;
283     }
284     other.top_ = 0;
285 
286 
287 #ifdef DEBUG
288 
289     assert(heap_store.is_valid(mark_store));
290     assert(other.heap_store.is_valid(other.mark_store));
291 
292 #endif
293 
294   }
295 
296   template <typename V, typename C, typename A, typename E,
297             typename H, typename M>
298   void
299   multiple_heap_framework<V, C, A, E, H, M>::swap(
300     multiple_heap_framework& other) {
301     // Precondition: The allocators and comparators are compatible.
302     heap_store.swap(other.heap_store);
303     mark_store.swap(other.mark_store);
304     std::swap(top_, other.top_);
305     std::swap(size_, other.size_);
```

```
306   }
307 }
308
309 #if defined(UNITTEST_MULTIPLE_HEAP_FRAMEWORK)
310 #include "blank-mark-store.h++"
311 #include "proxy-list-heap-store.h++"
312 #include "fat-weak-heap-node.h++"
313 #include <functional>
314 #include "lazy-mark-store.h++"
315 #include <memory>
316 #include "multiple-heap-framework.h++"
317 #include "pennant-node.h++"
318 #include "heap-proxy.h++"
319 #include "root-list-heap-store.h++"
320 #include "weak-heap-node.h++"
321
322 #include <algorithm>
323 #include <numeric>
324 #include <vector>
325
326 using namespace cphstl;
327
328 template <typename Q>
329 struct pqfw_test {
330   typedef typename Q::value_type V;
331   typedef typename Q::comparator_type C;
332   typedef typename Q::allocator_type A;
333   typedef typename Q::encapsulator_type E;
334   typedef typename Q::heap_store_type H;
335   typedef typename Q::mark_store_type M;
336
337   template <typename V, typename E, typename A>
338   static E* create(V const& v, A& allocator) {
339     E* p = allocator.allocate(1);
340     new (p) E(v, allocator);
341     return p;
342   }
343
344   template <typename V, typename E, typename A>
345   static void destroy(E* p, A& allocator) {
346     p->~E();
347     allocator.deallocate(p, 1);
348   }
349
350   static void small_test_multiple_heap_framework() {
351     Q w;
352
353     typedef typename A::template rebind<E>::other node_allocator_type;
354     node_allocator_type node_allocator;
355     E* p = create<V, E, node_allocator_type>(V(2), node_allocator);
356     E* q = create<V, E, node_allocator_type>(V(4), node_allocator);
357     E* r = create<V, E, node_allocator_type>(V(1), node_allocator);
358
359     std::cout << "insert\n";
360     w.insert(p);
361     assert(w.top() == p);
362     w.show();
363     assert(w.size() == 1);
364     std::cout << "insert " << (*q).element() << "\n";
365     w.insert(q);
366     assert(w.top() == q);
367     w.show();
368     assert(w.size() == 2);
369     std::cout << "extract\n";
370     E* s = w.extract();
```

```
371     w.show();
372     assert(w.size() == 1);
373     std::cout << "insert\n";
374     w.insert(r);
375     assert(w.top() != s && w.top() != r);
376     w.show();
377     assert(w.size() == 2);
378     w.extract();
379     w.extract();
380     assert(w.size() == 0);
381 
382     destroy<V, E, node_allocator_type>(p, node_allocator);
383     destroy<V, E, node_allocator_type>(q, node_allocator);
384     destroy<V, E, node_allocator_type>(r, node_allocator);
385   }
386 
387   static void big_test_multiple_heap_framework() {
388     Q h;
389     assert(h.size() == 0);
390     assert(h.begin() == h.end());
391     typename Q::allocator_type heap_a = h.get_allocator();
392     typename Q::comparator_type heap_c = h.get_comparator();
393     assert(h.size() <= h.max_size());
394     assert(h.top() == h.end());
395     V a[] = {8, 10, 13, 1, 6, 5, 3, 7, 11, 9};
396     unsigned int const n = sizeof(a) / sizeof(V);
397 
398     typedef typename A::template rebind<E>::other node_allocator_type;
399     node_allocator_type node_allocator;
400     std::vector<E*, node_allocator_type> node;
401     for (unsigned int j = 0; j < n; ++j) {
402       E* p = create<V, E, node_allocator_type>(a[j], node_allocator);
403       node.push_back(p);
404     }
405     h.insert(node[0]);
406     assert(h.top() == h.begin());
407     E* t = h.top();
408     assert((*t).element() == V(8));
409     for (unsigned int j = 1; j < n; ++j) {
410       h.insert(node[j]);
411       E* r = h.top();
412       assert((*r).element() == *(std::max_element(&a[0], &a[j + 1])));
413     }
414     assert(h.size() == n);
415 
416     V sum = V(0);
417     while (h.size() != 0) {
418       E* p =     h.extract();
419       sum += (*p).element();
420     }
421     assert(sum == std::accumulate(&a[0], &a[n], V(0)));
422 
423     Q g;
424     for (unsigned int j = 0; j < n; ++j) {
425       g.insert(node[j]);
426     }
427     assert(g.size() == n);
428 
429     std::sort(&a[0], &a[n]);
430     V another_sum = V(0);
431     while (g.size() != 0) {
432       E* p = g.top();
433       assert((*p).element() == a[g.size() - 1]);
434       another_sum += (*p).element();
435       g.extract(p);
```

```
436     }
437     assert(sum == another_sum);
438
439     for (unsigned int j = 0; j < n; ++j) {
440       destroy<V, E, node_allocator_type>(node[j], node_allocator);
441     }
442   }
443 };
444
445 template <typename V, typename C, typename A, typename E,
446   typename H, typename M>
447 void small_test_multiple_heap_framework() {
448     pqfw_test<multiple_heap_framework<V, C, A, E, H, M> >::
          small_test_multiple_heap_framework();
449 }
450
451 template <typename V, typename C, typename A, typename E,
452   typename H, typename M>
453 void big_test_multiple_heap_framework() {
454     pqfw_test<multiple_heap_framework<V, C, A, E, H, M> >::
          big_test_multiple_heap_framework();
455 }
456
457 int main(int, char**) {
458   typedef int V;
459   typedef std::less<V> C;
460   typedef std::allocator<V> A;
461
462   std::cout << "weak queues...\n";
463   typedef cphstl::weak_heap_node<V, A> E1; //<--
464   typedef cphstl::proxy_list_heap_store<C, A, E1> H1;
465   typedef cphstl::blank_mark_store<C, A, E1> M1;
466   small_test_multiple_heap_framework<V, C, A, E1, H1, M1>();
467   big_test_multiple_heap_framework<V, C, A, E1, H1, M1>();
468
469   std::cout << "weak queues with fat nodes...\n";
470   typedef cphstl::fat_weak_heap_node<V, A> E2; //<--
471   typedef cphstl::proxy_list_heap_store<C, A, E2> H2;
472   typedef cphstl::blank_mark_store<C, A, E2> M2; //<--
473   small_test_multiple_heap_framework<V, C, A, E2, H2, M2>();
474   big_test_multiple_heap_framework<V, C, A, E2, H2, M2>();
475
476   std::cout << "pennant queues...\n";
477   typedef cphstl::pennant_node<V, A> E3; //<--
478   typedef cphstl::proxy_list_heap_store<C, A, E3> H3;
479   typedef cphstl::blank_mark_store<C, A, E3> M3;
480   small_test_multiple_heap_framework<V, C, A, E3, H3, M3>();
481   big_test_multiple_heap_framework<V, C, A, E3, H3, M3>();
482
483   std::cout << "run-relaxed weak queues...\n";
484   typedef cphstl::fat_weak_heap_node<V, A> E4; //<--
485   typedef cphstl::proxy_list_heap_store<C, A, E4> H4;
486   typedef cphstl::lazy_mark_store<C, A, E4> M4; //<--
487   small_test_multiple_heap_framework<V, C, A, E4, H4, M4>();
488   big_test_multiple_heap_framework<V, C, A, E4, H4, M4>();
489
490   /*
491   typedef cphstl::weak_heap_node<V, A> E5; //<--
492   typedef cphstl::root_list_heap_store<C, A, E5> H5;
493   typedef cphstl::lazy_mark_store<C, A, E4> M5; //<--
494   small_test_multiple_heap_framework<V, C, A, E5, H5, M5>();
495   big_test_multiple_heap_framework<V, C, A, E5, H5, M5>();
496
497   */
498   return 0;
```

```
499 }
500
501 #endif
```

5. Heapifiers

5.1 Priority-queue-frameworks/Code/top-down-binary-heap-heapifier.h++

```
 1 /*
 2   A top-down binary-heap heapifier works as proposed by Williams in
 3   his seminal paper.
 4
 5   Authors: Stefan Edelkamp, Jyrki Katajainen © 2009, 2010
 6 */
 7
 8 #ifndef __CPHSTL_TOP_DOWN_BINARY_HEAP_HEAPIFIER__
 9 #define __CPHSTL_TOP_DOWN_BINARY_HEAP_HEAPIFIER__
10
11 #include <algorithm> // std::swap
12
13 namespace cphstl {
14
15   class top_down_binary_heap_heapifier {
16   public:
17
18     template <typename C, typename K, typename I>
19     void siftup(C const& comparator, K& vector, I j) {
20       I i = parent(j);
21       while ((j != 0) && comparator(vector[i]->element(), vector[j]->element())) {
22         std::swap(vector[i], vector[j]);
23         std::swap(vector[i]->position(), vector[j]->position());
24         j = i;
25         i = parent(j);
26       }
27     }
28
29     template <typename C, typename K, typename I>
30     I siftdown(C const& comparator, K& vector, I n, I i) {
31       I j = first_child(i);
32       while (j < n) {
33         if ((j + 1 < n)
34             && comparator(vector[j]->element(), vector[j + 1]->element())) {
35           j += 1;
36         }
37         if (comparator(vector[i]->element(), vector[j]->element())) {
38           std::swap(vector[i], vector[j]);
39           std::swap(vector[i]->position(), vector[j]->position());
40           i = j;
41           j = first_child(i);
42         }
43         else return i;
44       }
45       return i;
46     }
47
48 #ifdef DEBUG
49
50     template <typename C, typename K>
51     bool is_valid(C const& comparator, K& vector) {
52       bool okey = true;
53       for (typename K::size_type i = 1; i < vector.size(); ++i) {
54         if (comparator(vector[parent(i)]->element(), vector[i]->element())) {
55           std::cout << "error: " << vector[parent(i)]->element() << "&"
56                     << vector[i]->element() << std::endl;
```

```
57          okey = false;
58        }
59      }
60      return okey;
61    }
62
63 #endif
64
65   private:
66
67     template <typename I>
68     I parent(I j) {
69       return ((j - 1) / 2);
70     }
71
72     template <typename I>
73     I first_child(I j) {
74       return (2 * j + 1);
75     }
76   };
77 }
78
79 #endif
```

5.2 Priority-queue-frameworks/Code/weak-heap-heapifier.h++

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

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

```
104     template <typename I, typename K>
105     inline I second_child(I j, K& vector) {
106       return (2 * j + 1 - (vector[j]->position() < 0));
107     }
108   };
109 }
110
111 #endif
```

6. Encapsulators

6.1 Priority-queue-frameworks/Code/element-encapsulator.h++

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

6.2 Priority-queue-frameworks/Code/heap-node.h++

```
1  /*
2    A heap node used as a base for various specialized heap nodes
3
4    Authors: Asger Bruun, Jyrki Katajainen © 2009, 2010
5  */
6
7  #ifndef __CPHSTL_HEAP_NODE__
8  #define __CPHSTL_HEAP_NODE__
9
10 #include "assert.h++"
11 #include <cstddef> // std::size_t
12 #include <iostream>
13 #include <list>
14
15 namespace cphstl {
16
17   template <typename V, typename A, typename N>
18   class heap_node {
19   public:
20
21     typedef V value_type;
22     typedef A allocator_type;
23     typedef std::size_t size_type;
24     typedef unsigned char height_type;
25     typedef heap_node<V, A, N> self_type;
26
27     struct hole_type {
28       N* parent;
29       N* current;
30       union {
31         N* left;
32         void* owner;
33       };
34
35       hole_type(N* p)
36         : parent((*p).parent()), current(p) {
37         if (parent != 0) {
38           left = (*p).left();
39         }
40         else {
41           owner = (*p).owner();
42         }
43       }
44     };
45
46     N* parent_;
47     union {
48       N* left_;
49       void* owner_;
50     };
51     N* right_;
52     V value_;
53
54   private:
55
56     heap_node();
57     heap_node(heap_node const&);
58     heap_node& operator=(heap_node const&);
59
60   protected:
61
62     N const* const down_cast(self_type const* const b) const {
63       return static_cast<N const* const>(b);
64     }
65
```

```
66    N* const down_cast(self_type* const b) const {
67      return static_cast<N* const>(b);
68    }
69
70    N* const down_cast(self_type* const b) {
71      return static_cast<N* const>(b);
72    }
73
74  public:
75
76    heap_node(V const& v, A const&)
77      : parent_(down_cast(0)), left_(down_cast(0)), right_(down_cast(0)),
78      value_(v) {
79    }
80
81    static size_type footprint() {
82      return sizeof(N);
83    }
84
85    bool is_root() const {
86      return parent_ == 0;
87    }
88
89    bool is_leaf() const {
90      return right_ == 0;
91    }
92
93    V const& element() const {
94      return value_;
95    }
96
97    V& element() {
98      return value_;
99    }
100
101   N* left() const {
102     return left_;
103   }
104
105   N*& left() {
106     return left_;
107   }
108
109   N* right() const {
110     return right_;
111   }
112
113   N*& right() {
114     return right_;
115   }
116
117   N* parent() const {
118     return parent_;
119   }
120
121   N*& parent() {
122     return parent_;
123   }
124
125   void* owner() const {
126     return owner_;
127   }
128
129   void*& owner() {
130     return owner_;
```

```
131   }
132
133   template <typename C, typename M>
134   N* join(N* q, C const& comparator, M&) {
135     N* p = down_cast(this);
136     if (comparator((*p).element(), (*q).element())) {
137       N* c = (*q).right();
138       if (c != 0) {
139         (*c).parent() = p;
140       }
141       (*p).left() = c;
142       (*q).right() = p;
143       (*p).parent() = q;
144       return q;
145     }
146     else {
147       N* c = (*p).right();
148       if (c != 0) {
149         (*c).parent() = q;
150       }
151       (*q).left() = c;
152       (*p).right() = q;
153       (*q).parent() = p;
154       return p;
155     }
156   }
157
158   template <typename C, typename M>
159   N* fast_join(N* q, N*, C const& comparator, M& mark_store) {
160     N* p = down_cast(this);
161     return (*p).join(q, comparator, mark_store);
162   }
163
164   template <typename C>
165   N* split(C const&) {
166     N* p = down_cast(this);
167     assert(p != 0);
168     assert((*p).right() != 0);
169     N* q = (*p).right();
170     N* r = (*q).left();
171     (*p).right() = r;
172     if (r != 0) {
173       (*r).parent() = p;
174     }
175     (*q).parent() = 0;
176     (*q).left() = 0;
177     return q;
178   }
179
180   void swap_roots(N* q) {
181     N* p = down_cast(this);
182     assert((*p).is_root());
183     assert((*q).is_root());
184     N* c = (*p).right();
185     N* g = (*q).right();
186     (*p).right() = g;
187     (*q).right() = c;
188     if (c != 0) {
189       (*c).parent() = q;
190     }
191     if (g != 0) {
192       (*g).parent() = p;
193     }
194   }
195
```

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

```
261         N* g = (*q).right_;
262 
263         if (a != 0 && (*a).left_ == p) {
264           (*a).left_ = q;
265         }
266         if (a != 0 && (*a).right_ == p) {
267           (*a).right_ = q;
268         }
269         if (b != 0 && (! (*p).is_root()) && (*b).parent_ == p) {
270           (*b).parent_ = q;
271         }
272         if (c != 0 && (*c).parent_ == p) {
273           (*c).parent_ = q;
274         }
275         if (e != 0 && (*e).left_ == q) {
276           (*e).left_ = p;
277         }
278         if (e != 0 && (*e).right_ == q) {
279           (*e).right_ = p;
280         }
281         if (f != 0 && (! (*p).is_root()) && (*f).parent_ == q) {
282           (*f).parent_ = p;
283         }
284         if (g != 0 && (*g).parent_ == q) {
285           (*g).parent_ = p;
286         }
287 
288         (*p).parent_ = e;
289         (*p).left_ = f;
290         (*p).right_ = g;
291         (*q).parent_ = a;
292         (*q).left_ = b;
293         (*q).right_ = c;
294 
295         if (a == q && f == p) {
296           (*p).left_ = q;
297           (*q).parent_ = p;
298           return;
299         }
300         if (a == q && g == p) {
301           (*p).right_ = q;
302           (*q).parent_ = p;
303           return;
304         }
305         if (b == q) {
306           (*p).parent_ = q;
307           (*q).left_ = p;
308           return;
309         }
310         if (c == q) {
311           (*p).parent_ = q;
312           (*q).right_ = p;
313           return;
314         }
315       }
316 
317 #endif
318 
319       template <typename C>
320       N* distinguished_descendant(C const&) const {
321         N const* q = down_cast(this);
322         assert(q != 0);
323         return ((*q).is_root())? (*q).right() : (*q).left();
324       }
325 
```

```
326     N* distinguished_ancestor() const {
327       N const* q = down_cast(this);
328       assert(q != 0);
329       N* p = (*q).parent();
330       while (p != 0 && (*p).left() == q) {
331         q = p;
332         p = (*p).parent();
333       }
334       return p;
335     }
336
337     N* promote(N* p) {
338       N* q = down_cast(this);
339       assert(p == (*q).distinguished_ancestor());
340       if (p == (*q).parent()) {
341         assert((*p).right() == q);
342         N* a = (*p).parent();
343         N* b = (*p).left();
344         N* f = (*q).left();
345         N* g = (*q).right();
346         (*p).parent() = q;
347         (*p).left() = f;
348         (*p).right() = g;
349         (*q).parent() = a;
350         (*q).left() = b;
351         (*q).right() = p;
352         if (a != 0) {
353           if ((*a).right() == p) {
354             (*a).right() = q;
355           }
356           else {
357             if (! (*p).is_root()) {
358               (*a).left() = q;
359             }
360           }
361         }
362         if (b != 0 && (! (*p).is_root())) {
363           (*b).parent() = q;
364         }
365         if (f != 0) {
366           (*f).parent() = p;
367         }
368         if (g != 0) {
369           (*g).parent() = p;
370         }
371       }
372       else {
373         N* a = (*p).parent();
374         N* b = (*p).left();
375         N* c = (*p).right();
376         N* e = (*q).parent();
377         N* f = (*q).left();
378         N* g = (*q).right();
379         (*p).parent() = e;
380         (*p).left() = f;
381         (*p).right() = g;
382         (*q).parent() = a;
383         (*q).left() = b;
384         (*q).right() = c;
385         if (a != 0) {
386           if ((*a).right() == p) {
387             (*a).right() = q;
388           }
389           else {
390             if (! (*p).is_root()) {
```

```
391             (*a).left() = q;
392           }
393         }
394       }
395       if (b != 0 && (! (*p).is_root())) {
396         (*b).parent() = q;
397       }
398       if (c != 0) {
399         (*c).parent() = q;
400       }
401       if (e != 0 && (*e).left() == q) {
402         (*e).left() = p;
403       }
404       if (e != 0 && (*e).right() == q) {
405         (*e).right() = p;
406       }
407       if (f != 0) {
408         (*f).parent_ = p;
409       }
410       if (g != 0) {
411         (*g).parent_ = p;
412       }
413     }
414     return q;
415   }
416
417   N const* root() const {
418     N const* p = down_cast(this);
419     assert(p != 0);
420     while (! (*p).is_root()) {
421       p = (*p).parent();
422     }
423     return p;
424   }
425
426   N const* successor() const {
427     N const* x = down_cast(this);
428     assert(x != 0);
429     if ((*x).right() != 0) {
430       x = (*x).right();
431       while ((*x).left() != 0) {
432         x = (*x).left();
433       }
434       return x;
435     }
436     N const* y = (*x).parent();
437     while (y != 0 && x == (*y).right()) {
438       x = y;
439       y = (*y).parent();
440     }
441     return y;
442   }
443
444   height_type height() const {
445     N const* p = down_cast(this);
446     assert(p != 0);
447     height_type h = 0;
448     while (! (*p).is_leaf()) {
449       p = (*p).right();
450       h += 1;
451     }
452     return h;
453   }
454
455 #ifdef DEBUG
```

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

6.3 Priority-queue-frameworks/Code/weak-heap-node.h++

```
 1  /*
 2    A weak-heap node
 3
 4    Author: Jyrki Katajainen © 2009
 5  */
 6
 7  #ifndef __CPHSTL_WEAK_HEAP_NODE__
 8  #define __CPHSTL_WEAK_HEAP_NODE__
 9
10  #include "heap-node.h++"
11
12  namespace cphstl {
13
14    template <typename V, typename A>
15    class weak_heap_node
16      : public heap_node<V, A, weak_heap_node<V, A> > {
17
18    public:
19
20      typedef V value_type;
21      typedef A allocator_type;
22      typedef weak_heap_node<V, A> N;
23
```

```
24   private:
25
26     weak_heap_node();
27     weak_heap_node(N const&);
28     weak_heap_node& operator=(N const&);
29
30   public:
31
32     weak_heap_node(V const& v, A const& a)
33       : heap_node<V, A, N>(v, a) {
34     }
35
36 #ifdef DEBUG
37
38     template <typename C, typename M>
39     bool is_valid(C const& comparator, M const& mark_store) const {
40       N const* t = (*this).down_cast(this);
41       bool valid = true;
42       if ((*t).parent() != 0) {
43         valid &= t -> parent() -> left() == t ||
44           t -> parent() -> right() == t;
45         if (! valid) std::cout << "parent\n";
46       }
47       if (! (*t).is_root() && (*t).left() != 0) {
48         valid &= t -> left() -> parent() == t;
49         if (! valid) std::cout << "left\n";
50       }
51       if ((*t).right() != 0) {
52         valid &= t -> right() -> parent() == t;
53         if (! valid) {
54           std::cout << "right\n";
55           std::cout << "t: " << (*t).element() << " ";
56           std::cout << "right: " << t -> right() -> element() << " ";
57           std::cout << "up again: " << t -> right() -> parent() -> element() << "\n"
             ;
58         }
59       }
60       if (! (*t).is_root() && ! mark_store.is_marked(t)) {
61         valid &= ! comparator((*t).distinguished_ancestor() -> element(), (*t).
             element());
62
63         if (! valid) std::cout << "ancestor\n";}
64       return valid;
65     }
66
67 #endif
68
69   };
70 }
71
72 #endif
```

6.4 Priority-queue-frameworks/Code/fat-weak-heap-node.h++

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

```
12 #include "heap-node.h++"
13
14 namespace cphstl {
15
16   template <typename V, typename A>
17   class fat_weak_heap_node
18     : public heap_node<V, A, fat_weak_heap_node<V, A> > {
19   public:
20
21     typedef unsigned char height_type;
22     typedef signed char index_type;
23     enum mark_type {unmarked = 0, member, leader, singleton};
24     typedef fat_weak_heap_node<V, A> N;
25     typedef heap_node<V, A, N> B;
26
27   protected:
28
29     height_type height_;
30     index_type index_;
31     mark_type type_;
32
33     B const* const up_cast(N const* const d) const {
34       return static_cast<B const* const>(d);
35     }
36
37     B* const up_cast(N* const d) const {
38       return static_cast<B* const>(d);
39     }
40
41     B* const up_cast(N* const d) {
42       return static_cast<B* const>(d);
43     }
44
45   public:
46
47     fat_weak_heap_node(V const& v, A const& a)
48       : heap_node<V, A, N>(v, a), height_(0), index_(-1),
49         type_(unmarked) {
50     }
51
52     height_type height() const {
53       assert(this != 0);
54       return height_;
55     }
56
57     height_type& height() {
58       assert(this != 0);
59       return height_;
60     }
61
62     index_type index() const {
63       assert(this != 0);
64       return index_;
65     }
66
67     index_type& index() {
68       assert(this != 0);
69       return index_;
70     }
71
72     mark_type type() const {
73       assert(this != 0);
74       return type_;
75     }
76
```

```
77    mark_type& type() {
78      assert(this != 0);
79      return type_;
80    }
81
82    template <typename C>
83    N* basic_join(N* q, C const& comparator) {
84      N* p = this;
85      if (comparator((*p).element(), (*q).element())) {
86        N* c = (*q).right();
87        if (c != 0) {
88          (*c).parent() = p;
89        }
90        (*p).left() = c;
91        (*q).right() = p;
92        (*p).parent() = q;
93        (*q).height() += 1;
94        return q;
95      }
96      else {
97        N* c = (*p).right();
98        if (c != 0) {
99          (*c).parent() = q;
100       }
101       (*q).left() = c;
102       (*p).right() = q;
103       (*q).parent() = p;
104       (*p).height() += 1;
105       return p;
106     }
107   }
108
109   template <typename C, typename M>
110   N* join(N* q, C const& comparator, M& mark_store) {
111     N* p = this;
112     assert(p != 0);
113     assert(q != 0);
114     assert((*p).type() == N::unmarked);
115     assert((*q).type() == N::unmarked);
116     B* u = up_cast(p);
117     N* r = (*u).join(q, comparator, mark_store);
118     (*r).height() += 1;
119     N* a = (*r).right() -> left();
120     N* b = (*r).right() -> right();
121     bool a_unmarked = a == 0 || (*a).type() == N::unmarked;
122     bool b_unmarked = b == 0 || (*b).type() == N::unmarked;
123     int item = 2 * a_unmarked + b_unmarked;
124     switch (item) {
125     case 0 /* marked marked */:
126       (void) mark_store.sibling_transformation(a);
127       break;
128     case 1 /* marked unmarked */:
129       mark_store.cleaning_transformation(a);
130       break;
131     case 2 /* unmarked marked */:
132       break;
133     case 3 /* unmarked unmarked */:
134       break;
135     default:
136       assert(false);
137     }
138     return r;
139   }
140
141   template <typename C, typename M>
```

```
142     N* fast_join(N* q, N*, C const& comparator, M& mark_store) {
143       N* p = this;
144       assert(p != 0);
145       assert(q != 0);
146       assert((*p).type() == N::unmarked);
147       assert((*q).type() == N::unmarked);
148       assert((*p).right() == 0 || (*p).right() -> type() == N::unmarked);
149       if (comparator((*p).element(), (*q).element())) {
150         N* b = (*q).right();
151         bool cleaning_necessary = false;
152         if (b != 0) {
153           (*b).parent() = p;
154           if ((*b).type() != N::unmarked) {
155             cleaning_necessary = true;
156           }
157         }
158         (*p).left() = b;
159         (*q).right() = p;
160         (*p).parent() = q;
161         (*q).height() += 1;
162         if (cleaning_necessary) {
163           mark_store.cleaning_transformation(b);
164         }
165         return q;
166       }
167       else {
168         N* a = (*p).right();
169         if (a != 0) {
170           (*a).parent() = q;
171         }
172         (*q).left() = a;
173         (*p).right() = q;
174         (*q).parent() = p;
175         (*p).height() += 1;
176         return p;
177       }
178     }
179
180     template <typename C>
181     N* split(C const& comparator) {
182       assert((*this).height() > 0);
183       B* b = up_cast(this);
184       N* p = (*b).split(comparator);
185       (*this).height() -= 1;
186       return p;
187     }
188
189     void swap_neighbours(N* r) {
190       N* p = this;
191       assert(p != 0);
192       assert(r == (*p).right());
193       N* a = (*p).left();
194       N* b = (*r).left();
195       N* c = (*r).right();
196       (*r).left() = a;
197       if (a != 0 && (! (*p).is_root())) {
198         (*a).parent() = r;
199       }
200       (*r).right() = p;
201       (*p).parent() = r;
202       (*p).left() = c;
203       (*p).right() = b;
204       if (b != 0) {
205         (*c).parent() = p; // be careful!
206         (*b).parent() = p;
```

```
207       }
208       (*r).height() += 1;
209       (*p).height() -= 1;
210     }
211
212     void swap_roots(N* q) {
213       B* b = up_cast(this);
214       (*b).swap_roots(q);
215       std::swap((*this).height(), (*q).height());
216     }
217
218     N* release_root() {
219       B* b = up_cast(this);
220       N* s = (*b).release_root();
221       (*this).height() = 0;
222       return s;
223     }
224
225     N* promote(N* p) {
226       B* b = up_cast(this);
227       N* q = (*b).promote(p);
228       std::swap((*this).height(), (*p).height());
229       return q;
230     }
231
232 #ifdef DEBUG
233
234     void show_tree() const {
235       N const* t = this;
236       std::list<N const*> level;
237       level.push_front(t);
238       while (! level.empty()) {
239         typename std::list<N const*>::iterator last = level.end();
240         --last;
241         typename std::list<N const*>::iterator p = level.begin();
242         bool stop = false;
243         while (! stop) {
244           N const* t = *p;
245           if (p == last) {
246             stop = true;
247           }
248           ++p;
249           level.pop_front();
250           if (! (*t).is_root() && (*t).left() != 0) {
251             level.push_back((*t).left());
252           }
253           if ((*t).right() != 0) {
254             level.push_back((*t).right());
255           }
256           std::cout << "(" << (*t).element() <<
257             ", " << int((*t).height()) <<
258             ", " << int((*t).index()) <<
259             ", " << (*t).type() << ") ";
260         }
261         std::cout << "\n";
262         std::cout.flush();
263       }
264     }
265
266     template <typename C, typename M>
267     bool is_valid(C const& comparator, M const& mark_store) const {
268       N const* t = this;
269       bool valid = true;
270       if ((*t).parent() != 0) {
271         valid &= t -> parent() -> left() == t ||
```

```
272             t -> parent() -> right() == t;
273           valid &= t -> parent() -> height() == t -> height() + 1;
274           if (! valid) std::cout << "error: parent\n";
275         }
276         if ((*t).left() != 0) {
277           valid &= t -> left() -> parent() == t;
278           if (! valid) std::cout << "error: left\n";
279         }
280         if ((*t).right() != 0) {
281           valid &= t -> right() -> parent() == t;
282           if (! valid) std::cout << "error: right\n";
283         }
284         if (! (*t).is_root() && ! mark_store.is_marked(t)) {
285           valid &= ! comparator((*t).distinguished_ancestor() -> element(), (*t).
                  element());
286           if (! valid) std::cout << "error: half order\n";
287         }
288         if ((*t).is_root()) {
289           valid &= (*t).type() == N::unmarked;
290           assert((*t).parent() == 0);
291           if (! valid) std::cout << "error: root\n";
292         }
293         else {
294           N const* p = (*t).parent();
295           N const* r = (*t).left();
296           if ((*p).left() == t) {
297             if (mark_store.is_marked(p)) {
298               if (mark_store.is_marked(t)) {
299                 valid &= (*t).type() == N::member;
300                 if (! valid) std::cout << "error: member\n";
301               }
302             }
303             else {
304               if (mark_store.is_marked(t)) {
305                 if (r != 0 && mark_store.is_marked(r)) {
306                   valid &= (*t).type() == N::leader;
307                   if (! valid) std::cout << "error: leader (left)\n";
308                 }
309                 else {
310                   valid &= (*t).type() == N::singleton;
311                   if (! valid) std::cout << "error: singleton (left)\n";
312                 }
313               }
314             }
315           }
316           else {
317             if (r != 0 && mark_store.is_marked(r)) {
318               if (mark_store.is_marked(t)) {
319                 valid &= (*t).type() == N::leader;
320                 if (! valid) std::cout << "error: leader (right)\n";
321               }
322             }
323             else {
324               if (mark_store.is_marked(t)) {
325                 valid &= (*t).type() == N::singleton;
326                 if (! valid) std::cout << "error: singleton (right)\n";
327               }
328             }
329           }
330         }
331         return valid;
332       }
333
334   #endif
335
```

```
336   };
337 }
338
339 #endif
```

7. Heap store

7.1 Priority-queue-frameworks/Code/heap-proxy.h++

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

```
56        return height_;
57      }
58
59      heap_proxy* successor() const {
60        return next_;
61      }
62
63      heap_proxy*& successor() {
64        return next_;
65      }
66
67      heap_proxy* successor_pair() const {
68        return next_pair_;
69      }
70
71      heap_proxy*& successor_pair() {
72        return next_pair_;
73      }
74
75    protected:
76
77      E* root_;
78      size_type height_;
79      heap_proxy* next_;
80      heap_proxy* next_pair_;
81
82    private:
83
84      heap_proxy();
85      heap_proxy(heap_proxy const&);
86      heap_proxy& operator=(heap_proxy const&);
87
88    };
89  }
90
91  #if defined(UNITTEST_HEAP_PROXY)
92
93  #include <memory> // std::allocator
94  #include "weak-heap-node.h++"
95
96  template <typename T>
97  void test_heap_proxy() {
98    typedef std::allocator<T> A;
99    typedef cphstl::weak_heap_node<T, A> N;
100   typedef cphstl::heap_proxy<N> P;
101   N* dummy = new N(T(0), A());
102   P* p = new P(dummy);
103   P* q = new P(dummy);
104   P* r = new P(dummy);
105
106   (*p).successor() = q;
107   (*q).successor() = r;
108   (*r).successor() = 0;
109
110   assert((*p).height() == 0);
111   assert((*q).root() == dummy);
112   assert((*r).successor_pair() == 0);
113   assert((*p).successor() == q);
114   assert((*q).successor() == r);
115   assert((*r).successor() == 0);
116
117   (*r).height() = 4;
118   assert((*r).height() == 4);
119 }
120
```

```
121 int main(int, char**) {
122   test_heap_proxy<int>();
123   test_heap_proxy<char>();
124   return 0;
125 }
126
127 #endif
128 #endif
```

7.2 Priority-queue-frameworks/Code/proxy-list-heap-store.h++

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

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

```
120         E* q = (*p).join((*head).root(), comparator, mark_store);
121         (*head).update(q, (*head).height() + 1, (*head).successor(), 0);
122         P* s = (*head).successor();
123         if ((s != 0) && ((*head).height() == (*s).height())) {
124           first_pair = head;
125         }
126         return;
127       }
128       P* u = create(p, h, 0, 0);
129       (*u).successor() = head;
130       head = u;
131     }
132 
133     void inject_without_join(E* p, size_type h) {
134       assert((*p).is_root());
135       number_of_nodes += 1 << h;
136       P* u = create(p, h, 0, 0);
137       (*u).successor() = head;
138       head = u;
139     }
140 
141     std::pair<E*, size_type> eject() {
142       // Warning: top_ can point to the nodes ejected
143       if (head == 0) {
144         return std::make_pair((E*) 0, (size_type) 0);
145       }
146       P* f = head;
147       E* r = (*f).root();
148       head = (*head).successor();
149       if (first_pair == f) {
150         first_pair = (*f).successor_pair();
151         (*f).successor_pair() = 0;
152       }
153       number_of_nodes -= 1 << (*f).height();
154       std::pair<E*, size_type> answer = std::make_pair(r, (*f).height());
155       destroy(f);
156       (*r).owner() = (P*) 0;
157       return answer;
158     }
159 
160     template <typename M>
161     void concatenate(E* q, size_type h, M& mark_store) {
162       while (q != 0) {
163         mark_store.unmark(q);
164         E* s = (*q).release_subheap();
165         inject_without_join(q, h);
166         h = h - 1;
167         q = s;
168       }
169     }
170 
171     void replace(E* new_root) {
172       heap_proxy_type* proxy = (heap_proxy_type*) (*new_root).owner();
173       (*proxy).root() = new_root;
174       (*new_root).owner() = (void *) proxy;
175     }
176 
177     template <typename M>
178     void meld(proxy_list_heap_store& other, M& mark_store) {
179       // Precondition: The comparators and allocators must be compatible.
180       std::list<P*, A> tmp;
181       size_type n = number_of_nodes;
182       size_type m = other.number_of_nodes;
183       P* s = begin();
184       P* t = other.begin();
```

```
185       while (s != 0 && t != 0) {
186         if ((*s).height() < (*t).height()) {
187           tmp.push_front(s);
188           s = (*s).successor();
189           pop();
190         }
191         else {
192           tmp.push_front(t);
193           t = (*t).successor();
194           other.pop();
195         }
196       }
197       if (begin() == 0) {
198         swap(other);
199       }
200       while (! tmp.empty()) {
201         P* t = tmp.front();
202         E* r = (*t).root();
203         inject(r, (*t).height(), mark_store);
204         destroy(t);
205         tmp.pop_front();
206       }
207       number_of_nodes = n + m;
208       other.number_of_nodes = 0;
209     }
210
211     void swap(proxy_list_heap_store& other) {
212       // Precondition: The comparators and allocators must be compatible.
213       std::swap(head, other.head);
214       std::swap(first_pair, other.first_pair);
215       std::swap(number_of_nodes, other.number_of_nodes);
216     }
217
218 #ifdef DEBUG
219
220     void show() {
221       std::cout << "========\n";
222       for (P* p = begin(); p != 0; p = (*p).successor()) {
223         std::cout << "--- " << (*p).height() << "\n";
224         E* r = (*p).root();
225         (*r).show_tree();
226       }
227       std::cout << "========\n";
228       std::cout.flush();
229     }
230
231     template <typename M>
232     bool is_valid(M const& mark_store) {
233       P const* p = begin();
234       if (p == 0) {
235         return true;
236       }
237       for (; p != 0; p = (*p).successor()) {
238         bool valid = true;
239         E const* r = (*p).root();
240         for (; r != 0; r = (*r).successor()) {
241           if (! (*r).is_valid(comparator, mark_store)) {
242             std::cerr << "heap-structure error: " << (*r).element() << "\n";
243             valid = false;
244           }
245         }
246         if (! valid) {
247           return false;
248         }
249       }
```

```
250       p = begin();
251       P const* q = (*p).successor();
252       for (; q != 0; p = q, q = (*p).successor()) {
253         if ((*p).height() > (*q).height()) {
254           std::cerr << "height violation: " << (*p).height() << "\n";
255           return false;
256         }
257       }
258       p = begin();
259       q = (*p).successor();
260       for (; q != 0; p = q, q = (*p).successor()) {
261         if ((*p).successor_pair() != 0 && (*p).height() != (*q).height()) {
262           std::cerr << "pair violation: " << (*p).height() << "\n";
263           return false;
264         }
265       }
266       p = begin();
267       q = (*p).successor();
268       for (; q != 0; p = q, q = (*p).successor()) {
269         E const* r = (*p).root();
270         if (! (*r).is_root()) {
271           std::cerr << "root violation: " << (*p).height() << "\n";
272           return false;
273         }
274         if ((*r).owner() != p) {
275           std::cerr << "owner violation: " << (*p).height() << "\n";
276           return false;
277         }
278       }
279       return true;
280     }
281 
282 #endif
283 
284   protected:
285 
286     P* create(E* r, size_type height, P* next, P* next_pair) {
287       assert((*r).is_root());
288       P* u = proxy_allocator.allocate(1);
289       new (u) P(r, height, next, next_pair);
290       (*r).owner() = u;
291       return u;
292     }
293 
294     void destroy(P* u) {
295       assert(u != 0);
296       (*u).~P();
297       proxy_allocator.deallocate(u, 1);
298     }
299 
300     void pop() {
301       P* f = head;
302       head = (*head).successor();
303       if (first_pair == f) {
304         first_pair = (*f).successor_pair();
305         (*f).successor_pair() = 0;
306       }
307     }
308   };
309 }
310 
311 #if defined(UNITTEST_PROXY_LIST_HEAP_STORE)
312 
313 #include "blank-mark-store.h++"
314 #include <cstddef> // std::size_t
```

```
315 #include <functional>
316 #include <iostream>
317 #include <memory>
318 #include <numeric> // std::accumulate
319 #include "weak-heap-node.h++"
320
321 template <typename E, typename H, typename M>
322 void push(E* p, H& heap_store, M& mark_store) {
323   heap_store.inject(p, 0, mark_store);
324 }
325
326 template <typename E, typename C, typename H, typename M>
327 E* extract(C const& comparator, H& heap_store, M& mark_store) {
328   typedef std::size_t size_type;
329   std::pair<E*, size_type> pair = heap_store.eject();
330   E* r = pair.first;
331   size_type h = pair.second;
332   while ((*r).right() != 0) {
333     E* q = (*r).split(comparator);
334     h = h - 1;
335     heap_store.inject(q, h, mark_store);
336   }
337   return r;
338 }
339
340 template <typename T>
341 void proxy_list_heap_store_test() {
342   typedef std::allocator<T> A;
343   typedef std::less<T> C;
344   typedef cphstl::weak_heap_node<T, A> E;
345   typedef cphstl::heap_proxy<E> P;
346   typedef cphstl::proxy_list_heap_store<C, A, E, P> H;
347   typedef cphstl::blank_mark_store<C, A, E> M;
348
349   A allocator;
350   C less;
351   H heap_store(less, allocator);
352   M mark_store(less, allocator);
353   E m(T(4), allocator);
354   E n(T(2), allocator);
355   E o(T(3), allocator);
356   E* p = new E(T(1), allocator);
357
358   std::cout << "push ";
359   push(&m, heap_store, mark_store);
360   heap_store.show();
361   std::cout << "push ";
362   push(&n, heap_store, mark_store);
363   heap_store.show();
364   std::cout << "push ";
365   push(&o, heap_store, mark_store);
366   heap_store.show();
367   std::cout << "push ";
368   push(p, heap_store, mark_store);
369   heap_store.show();
370   assert(heap_store.is_valid(mark_store));
371   std::cout << "validated\n";
372
373   std::cout << "extract ";
374   E* r = extract<E, C, H>(less, heap_store, mark_store);
375   heap_store.show();
376
377   std::cout << "push ";
378   push(r, heap_store, mark_store);
379   assert(heap_store.is_valid(mark_store));
```

```
380   std::cout << "validated\n";
381 }
382
383 template <typename T>
384 void iteration_test() {
385   typedef std::allocator<T> A;
386   typedef std::less<T> C;
387   typedef cphstl::weak_heap_node<T, A> E;
388   typedef cphstl::heap_proxy<E> P;
389   typedef cphstl::proxy_list_heap_store<C, A, E, P> H;
390   typedef cphstl::blank_mark_store<C, A, E> M;
391
392   A allocator;
393   C less;
394   T a[] = {8, 10, 12, 1, 6, 5, 3, 7};
395   unsigned int k = sizeof(a) / sizeof(T);
396   H heap_store(less, allocator);
397   M mark_store(less, allocator);
398   for (unsigned int j = 0; j < k; ++j) {
399     E* r = new E(a[j], allocator);
400     push(r, heap_store, mark_store);
401   }
402
403   T sum = T(0);
404   for (P const* h = heap_store.begin(); h != 0; h = (*h).successor()) {
405     for (E const* p = (*h).root(); p != 0; p = (*p).successor()) {
406       sum += (*p).element();
407     }
408   }
409   assert(sum == std::accumulate(&a[0], &a[k], T(0)));
410 }
411
412 int main(int, char**) {
413   proxy_list_heap_store_test<int>();
414   iteration_test<int>();
415   return 0;
416 }
417
418 #endif
419 #endif
```

8. Mark stores

8.1 Priority-queue-frameworks/Code/blank-mark-store.h++

```
 1 /*
 2   A blank mark store; a mark must be removed before any new marks are
 3   introduced.
 4
 5   Author: Jyrki Katajainen © 2009
 6 */
 7
 8 #ifndef __CPHSTL_BLANK_MARK_STORE__
 9 #define __CPHSTL_BLANK_MARK_STORE__
10
11 #include <algorithm> // std::swap
12 #include "assert.h++"
13 #include "comparator-proxy.h++"
14 #include <cstddef> // std::size_t
15 #include <iostream>
16
17 namespace cphstl {
18
19   template <typename C, typename A, typename E>
```

```
20   class blank_mark_store {
21   public:
22
23     typedef C comparator_type;
24     typedef A allocator_type;
25     typedef E encapsulator_type;
26     typedef std::size_t size_type;
27
28   protected:
29
30     comparator_proxy<C> comparator;
31     E* single_mark;
32
33   private:
34
35     blank_mark_store(blank_mark_store const&);
36     blank_mark_store& operator=(blank_mark_store const&);
37
38   public:
39
40     blank_mark_store(C const& c = C(), A const& = A())
41       : comparator(c), single_mark(0) {
42     }
43
44     ~blank_mark_store() {
45     }
46
47     size_type footprint(size_type) const {
48       return sizeof(blank_mark_store);
49     }
50
51     E* find_top() const {
52       return single_mark;
53     }
54
55     bool is_marked(E const* p) const {
56       return single_mark == p;
57     }
58
59     void mark(E* p) {
60       assert(single_mark == 0);
61       single_mark = p;
62     }
63
64     void unmark(E* p) {
65       single_mark = (single_mark == p) ? 0 : single_mark;
66     }
67
68     template <typename H>
69     void reduce(H& heap_store) {
70       if (single_mark == 0) {
71         return;
72       }
73       E* q = single_mark;
74       E* p = (*q).distinguished_ancestor();
75       while (p != 0) {
76         if (comparator((*p).element(), (*q).element())) {
77
78 #ifdef FIRST_VERSION
79
80           (*q).swap_nodes(p);
81
82 #else
83
84           q = (*q).promote(p);
```

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

8.2 Priority-queue-frameworks/Code/lazy-mark-store.h++

```
1 /*
2   A lazy mark store
3
4   Author: Jyrki Katajainen © 2009, 2010
5 */
6
7 #ifndef __CPHSTL_LAZY_MARK_STORE__
8 #define __CPHSTL_LAZY_MARK_STORE__
9
10 #include <algorithm> // std::max, std::swap
11 #include "assert.h++"
12 #include "bit-store.h++"
13 #include <cstddef> // std::size_t
14 #include <cmath> // ilogb
15 #include <iostream>
16 #include <vector>
```

```
17
18 extern int ilogb(double) throw();
19
20 namespace cphstl {
21
22   template <typename C, typename A, typename E>
23   class lazy_mark_store {
24   public:
25
26     typedef C comparator_type;
27     typedef A allocator_type;
28     typedef E encapsulator_type;
29     typedef typename E::mark_type mark_type;
30     typedef typename E::height_type height_type;
31     typedef typename E::index_type index_type;
32     typedef unsigned long word_type;
33     typedef std::size_t size_type;
34
35   protected:
36
37     C comparator;
38
39     bit_store<word_type> runs;
40     bit_store<word_type> teams;
41
42 #ifndef RANK
43
44     std::vector<E*, A> nodes;
45     std::vector<bit_store<word_type>, A> singles;
46
47 #endif
48
49   private:
50
51     lazy_mark_store(lazy_mark_store const&);
52     lazy_mark_store& operator=(lazy_mark_store const&);
53
54   public:
55
56     lazy_mark_store(C const& c = C(), A const& a = A())
57       : comparator(c), runs(), teams()
58
59 #ifndef RANK
60
61       , nodes(a)
62       , singles(a)
63
64 #endif
65     {
66     }
67
68     ~lazy_mark_store() {
69     }
70
71     size_type footprint(size_type n) const {
72       return (ilogb(n) + 1) * 6 * (sizeof(E*) + 2 * sizeof(word_type)) +
73         2 * sizeof(word_type);
74     }
75
76     E* find_top() const {
77
78 #ifndef RANK
79
80       if (nodes.size() == 0) {
81         return 0;
```

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

```
147           break;
148         case 2:
149           mark_action_2(p, q, r);
150           break;
151         case 3:
152           mark_action_2(p, q, r);
153           break;
154         case 4:
155           mark_action_1(p, q, r);
156           break;
157         case 5:
158           mark_action_1(p, q, r);
159           break;
160         case 6:
161           mark_action_2(p, q, r);
162           break;
163         case 7:
164           mark_action_2(p, q, r);
165           break;
166         case 8:
167           mark_action_3(p, q, r);
168           break;
169         case 9:
170           mark_action_3(p, q, r);
171           break;
172         case 10:
173           mark_action_4(p, q, r);
174           break;
175         case 11:
176           mark_action_4(p, q, r);
177           break;
178         case 12:
179           mark_action_5(p, q, r);
180           break;
181         case 13:
182           mark_action_5(p, q, r);
183           break;
184         case 14:
185           mark_action_6(p, q, r);
186           break;
187         case 15:
188           mark_action_6(p, q, r);
189           break;
190         default:
191           assert(false);
192         }
193       }
194
195       void unmark(E* q) {
196         //            std::cout << " unmark called " << q << std::endl;
197         assert(q != 0);
198         E* p = (*q).parent();
199         E* r = (*q).left();
200         E* s;
201         switch ((*q).type()) {
202         case E::unmarked:
203           return;
204         case E::member:
205           if (r == 0 || (*r).type() == E::unmarked) {
206             if ((*p).type() == E::member) {
207               // std::cout << "-- unmark action " << 1 << std::endl;
208               unmark_action_1(p, q, r);
209             }
210             else {
211               // std::cout << "-- unmark action " << 2 << std::endl;
```

```
212           unmark_action_2(p, q, r);
213         }
214       }
215       else {
216         assert((*r).type() == E::member);
217         E* s = (*r).left();
218         word_type row = word_type((*p).type() == E::leader);
219         word_type column = 0;
220         if (s != 0) {
221           if ((*s).type() == E::unmarked) {
222             column = 1;
223           }
224           else {
225             assert((*s).type() == E::member);
226             column = 2;
227           }
228         }
229         //  std::cout << "-- unmark case " << row*3+column << std::endl;
230         switch (row * 3 + column) {
231         case 0:
232           unmark_action_3(p, q, r);
233           break;
234         case 1:
235           unmark_action_3(p, q, r);
236           break;
237         case 2:
238           unmark_action_4(p, q, r);
239           break;
240         case 3:
241           unmark_action_5(p, q, r);
242           break;
243         case 4:
244           unmark_action_5(p, q, r);
245           break;
246         case 5:
247           unmark_action_6(p, q, r);
248           break;
249         default:
250           assert(false);
251         }
252       }
253       break;
254     case E::leader:
255       s = (*r).left();
256       if (s == 0) {
257         //    std::cout << "-- unmark action " << 7 << std::endl;
258         unmark_action_7(p, q, r);
259       }
260       else {
261         if ((*s).type() == E::unmarked) {
262           //        std::cout << "-- unmark action " << 7 << std::endl;
263           unmark_action_7(p, q, r);
264         }
265         else {        std::cout << "-- unmark action " << 8 << std::endl;
266           //
267           assert((*s).type() == E::member);
268           unmark_action_8(p, q, r);
269         }
270       }
271       break;
272     case E::singleton:
273       // std::cout << "-- unmark action " << 9 << std::endl;
274       unmark_action_9(p, q, r);
275       break;
276     default:
```

```
277         assert(false);
278       }
279         // std::cout << "end unmark  " << std::endl;
280     }
281 
282     template <typename H>
283     void reduce(H& heap_store) {
284 
285 #ifndef RANK
286       //      std::cout << " base reduce called "<< std::endl;
287       assert(is_valid());
288       if (teams.size() != 0) {
289         height_type h = teams.choose();
290         index_type i = singles[h].choose();
291         E* p = nodes[i];
292         assert((*p).type() == E::singleton);
293         singles[h].unset(i);
294         index_type j = singles[h].choose();
295         singles[h].set(i);
296         E* q = nodes[j];
297         assert((*q).type() == E::singleton);
298         singleton_transformation(p, q, heap_store);
299         return;
300       }
301       if (runs.size() != 0) {
302         index_type i = runs.choose();
303         E* r = nodes[i];
304         assert((*r).type() == E::leader);
305         run_transformation(r, heap_store);
306       }
307 
308 #endif
309     }
310 
311     template <typename H>
312     void meld(lazy_mark_store& other, H& heap_store) {
313 
314 #ifndef RANK
315 
316     index_type k = other.nodes.size();
317       for (index_type i = 0; i != k; ++i) {
318         E* p = other.nodes[i];
319         other.remove_mark(p);
320         other.remove_node(p);
321         insert_node(p);
322         insert_mark(p);
323         reduce(heap_store);
324       }
325       other.nodes.resize(0);
326       other.singles.resize(0);
327 
328 #endif
329     }
330 
331     void swap(lazy_mark_store& other) {
332 
333 #ifndef RANK
334 
335       // Precondition: The comparators are compatible.
336       std::swap(nodes, other.nodes);
337       std::swap(runs, other.runs);
338       std::swap(singles, other.singles);
339       std::swap(teams, other.teams);
340 
341 #endif
```

```
342     }
343
344 #ifdef DEBUG
345
346     bool is_valid() const {
347       bool valid = true;
348       for (index_type i = 0; i < index_type(nodes.size()); ++i) {
349         E const* p = nodes[i];
350         valid &= (*p).index() == i;
351         valid &= is_marked(p);
352         if (! valid) {
353           std::cout << "error: nodes\n";
354         }
355       }
356       bit_store<word_type> temp = runs;
357       for (height_type h = 0; h < height_type(temp.size()); ++h) {
358         index_type i = temp.choose();
359         temp.unset(i);
360         E const* p = nodes[i];
361         valid &= (*p).type() == E::leader;
362         if (! valid) {
363           std::cout << "error: runs\n";
364         }
365       }
366       temp = teams;
367       for (height_type j = 0; j < height_type(temp.size()); ++j) {
368         height_type h = temp.choose();
369         temp.unset(h);
370         valid &= singles[h].size() > 1;
371         if (! valid) {
372           std::cout << "error: teams\n";
373         }
374       }
375       for (height_type h = 0; h < height_type(singles.size()); ++h) {
376         temp = singles[h];
377         for (index_type i = 0; i < index_type(temp.size()); ++i) {
378           index_type j = temp.choose();
379           temp.unset(j);
380           E const* p = nodes[j];
381           valid &= (*p).type() == E::singleton;
382           if (! valid) {
383             std::cout << "error: singles (" << (*p).element() << ")\n";
384           }
385         }
386       }
387
388       return valid;
389     }
390
391 #endif
392
393   protected:
394
395     virtual void insert_node(E* q) {
396
397 #ifndef RANK // avoids index manip
398           // std::cout << "inserting base " << "/" << (*q).height() + 1 << std::
                endl;
399       nodes.push_back(q);
400       size_type max = std::max(singles.size(), size_type((*q).height() + 1));
401       singles.resize(max, bit_store<word_type>());
402       (*q).index() = nodes.size() - 1;
403
404 #endif
405     }
```

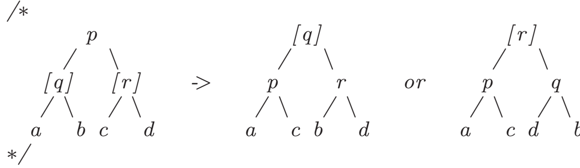
```
406
407     virtual void insert_mark(E* q) {
408
409 #ifndef RANK // avoids index manip
410
411       // Precondition: (*q).type() must have the new value
412       switch ((*q).type()) {
413       case E::leader:
414         runs.set((*q).index());
415         break;
416       case E::singleton:
417         singles[(*q).height()].set((*q).index());
418         if (singles[(*q).height()].size() > 1) {
419           teams.set((*q).height());
420         }
421         break;
422       default:
423         ;      }
424
425 #endif
426     }
427
428     virtual void remove_mark(E* r) {
429
430 #ifndef RANK
431       // Precondition: (*r).type() must have the old value
432       switch ((*r).type()) {
433       case E::leader:
434         runs.unset((*r).index());
435         break;
436       case E::singleton:
437         singles[(*r).height()].unset((*r).index());
438         if (singles[(*r).height()].size() < 2) {
439           teams.unset((*r).height());
440         }
441         break;
442       default:
443         ;
444       }
445
446 #endif
447     }
448
449     virtual void remove_node(E* x) {
450
451 #ifndef RANK
452
453       E* y = nodes.back();
454       remove_mark(y);
455       nodes.pop_back();
456       if (x != y) {
457         index_type i = (*x).index();
458         nodes[i] = y;
459         (*y).index() = i;
460         insert_mark(y);
461       }
462       assert(is_valid());
463
464 #endif
465     }
466
467
468     void mark_action_1(E*, E* q, E*) {
469       (*q).type() = E::singleton;
470       insert_node(q);
```

```
471     insert_mark(q);
472   }
473
474   void mark_action_2(E*, E* q, E* r) {
475     (*q).type() = E::leader;
476     insert_node(q);
477     insert_mark(q);
478     remove_mark(r);
479     (*r).type() = E::member;
480   }
481
482   void mark_action_3(E*, E* q, E*) {
483     (*q).type() = E::member;
484     insert_node(q);
485   }
486
487   void mark_action_4(E*, E* q, E* r) {
488     (*q).type() = E::member;
489     insert_node(q);
490     remove_mark(r);
491     (*r).type() = E::member;
492   }
493
494   void mark_action_5(E* p, E* q, E*) {
495     remove_mark(p);
496     (*p).type() = E::leader;
497     insert_mark(p);
498     (*q).type() = E::member;
499     insert_node(q);
500   }
501
502   void mark_action_6(E* p, E* q, E* r) {
503     remove_mark(p);
504     (*p).type() = E::leader;
505     insert_mark(p);
506     (*q).type() = E::member;
507     insert_node(q);
508     remove_mark(r);
509     (*r).type() = E::member;
510   }
511
512   void unmark_action_1(E*, E* q, E*) {
513     remove_node(q);
514     (*q).type() = E::unmarked;
515   }
516
517   void unmark_action_2(E* p, E* q, E*) {
518     remove_mark(p);
519     (*p).type() = E::singleton;
520     insert_mark(p);
521     remove_node(q);
522     (*q).type() = E::unmarked;
523   }
524
525   void unmark_action_3(E*, E* q, E* r) {
526     remove_node(q);
527     (*q).type() = E::unmarked;
528     (*r).type() = E::singleton;
529     insert_mark(r);
530   }
531
532   void unmark_action_4(E*, E* q, E* r) {
533     remove_node(q);
534     (*q).type() = E::unmarked;
535     (*r).type() = E::leader;
```

```
536      insert_mark(r);
537    }
538
539    void unmark_action_5(E* p, E* q, E* r) {
540      remove_mark(p);
541      (*p).type() = E::singleton;
542      insert_mark(p);
543      remove_node(q);
544      (*q).type() = E::unmarked;
545      (*r).type() = E::singleton;
546      insert_mark(r);
547    }
548
549    void unmark_action_6(E* p, E* q, E* r) {
550      remove_mark(p);
551      (*p).type() = E::singleton;
552      insert_mark(p);
553      remove_node(q);
554      (*q).type() = E::unmarked;
555      (*r).type() = E::leader;
556      insert_mark(r);
557    }
558
559    void unmark_action_7(E*, E* q, E* r) {
560      remove_mark(q);
561      remove_node(q);
562      (*q).type() = E::unmarked;
563      (*r).type() = E::singleton;
564      insert_mark(r);
565    }
566
567    void unmark_action_8(E*, E* q, E* r) {
568      remove_mark(q);
569      remove_node(q);
570      (*q).type() = E::unmarked;
571      (*r).type() = E::leader;
572      insert_mark(r);
573      assert(is_valid());
574    }
575
576    void unmark_action_9(E*, E* q, E*) {
577      remove_mark(q);
578      remove_node(q);
579      (*q).type() = E::unmarked;
580    }
581
582    template <typename H>
583    void cleanparent_transformation(E* q, H& heap_store) {
584       // std::cout << " cleanparent trans " << q  << std::endl;
585      assert(is_marked(q));
586      E* p = (*q).parent();
587      assert((*p).type() == E::unmarked);
588      assert((*p).left() == q);
589      E* r = (*p).right();
590      assert((*r).type() == E::unmarked);
591      unmark(q);
592      (*p).left() = r;
593      (*p).right() = q;
594      E* a = (*q).left();
595      E* c = (*r).left();
596      (*q).left() = c;
597      (*r).left() = a;
598      if (a != 0) {
599        (*a).parent() = r;
600      }
```

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

```
666
667   /*
```




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

```
731       (*p).left() = a;
732       (*p).right() = c;
733       (*r).left() = b;
734       (*r).right() = d;
735       if (a != 0) {
736         (*a).parent() = p;
737         (*c).parent() = p;
738         (*b).parent() = r;
739         (*d).parent() = r;
740       }
741       mark(q);
742       assert(is_valid());
743       return q;
744     }
745
746   protected:
747
748     /*
749         (p)                   (p)                   [q]
750        /   \                 /   \                 /   \
751       a     [q]     ->      a     q      or       a     p
752             / \                  / \                   / \
753            b   c                b   c                 c   b
754     */
755
756     template <typename H>
757     void parent_transformation(E* q, H& heap_store) {
758        // std::cout << " parent trans " << q << std::endl;
759       assert(is_marked(q));
760       E* p = (*q).parent();
761       assert((*p).right() == q);
762       assert(is_valid());
763       if (is_marked(p)) {
764         unmark(p);
765         unmark(q);
766         typename E::hole_type hole_at_p = (*p).splice_out();
767         q = (*p).split(comparator);
768         E* r = (*p).basic_join(q, comparator);
769         (*r).splice_in(hole_at_p, heap_store);
770         mark(r);
771       }
772       else {
773         unmark(q);
774         typename E::hole_type hole_at_p = (*p).splice_out();
775         (void) (*p).split(comparator);
776         E* r = (*p).basic_join(q, comparator);
777         (*r).splice_in(hole_at_p, heap_store);
778         if (q == r) {
779           mark(r);
780         }
781       }
782       assert(is_valid());
783     }
784
785     /*
786       p          r                p          r        [q]        [s]
787        \          \                \          \         \          \
788        [q]   ,    [s]    ->         r    or    p   ,     s    or    q
789        / \        / \              / \        / \       / \        / \
790       a   b      c   d            a   c      c   a     b   d      d   b
791     */
792
793     template <typename H>
794     void pair_transformation(E* q, E* s, H& heap_store) {
795        // std::cout << " pair trans " << q << "," << s << std::endl;
```

```
796     assert(q != s);
797     assert((*q).height() == (*s).height());
798     assert(! (*q).is_root() && ! (*s).is_root());
799     E* p = (*q).parent();
800     assert(q == (*p).right());
801     unmark(q);
802     E* r = (*s).parent();
803     assert(s == (*r).right());
804     unmark(s);
805     typename E::hole_type hole_at_p = (*p).splice_out();
806     typename E::hole_type hole_at_r = (*r).splice_out();
807     (void) (*p).split(comparator);
808     (void) (*r).split(comparator);
809     E* t = (*p).basic_join(r, comparator);
810     E* u = (*q).basic_join(s, comparator);
811     if (p == t) {
812       (*t).splice_in(hole_at_p, heap_store);
813       (*u).splice_in(hole_at_r, heap_store);
814     }
815     else {
816       (*t).splice_in(hole_at_r, heap_store);
817       (*u).splice_in(hole_at_p, heap_store);
818     }
819     mark(u);
820     assert(is_valid());
821   }
822
823   template <typename H>
824   void singleton_transformation(E* q, E* s, H& heap_store) {
825      // std::cout << " singleton trans " << q << "," << s << std::endl;
826     assert(is_marked(q));
827     assert(is_marked(s));
828     E* p = (*q).parent();
829     if ((*p).right() == q) {
830       if (is_marked(p)) {
831         parent_transformation(q, heap_store);
832         return;
833       }
834       if (! (*p).is_root() && is_marked((*p).left())) {
835         sibling_transformation((*p).left());
836         return;
837       }
838     }
839     else {
840       if (is_marked((*p).right())) {
841         if (is_marked(p)) {
842           parent_transformation((*p).right(), heap_store);
843           return;
844         }
845         sibling_transformation(q);
846         return;
847       }
848       cleaning_transformation(q);
849     }
850     E* r = (*s).parent();
851     if ((*r).right() == s) {
852       if (is_marked(r)) {
853         parent_transformation(s, heap_store);
854         return;
855       }
856       if (! (*r).is_root() && is_marked((*r).left())) {
857         sibling_transformation((*r).left());
858         return;
859       }
860     }
```

```
861       else {
862         if (is_marked((*r).right())) {
863           if (is_marked(r)) {
864             parent_transformation((*r).right(), heap_store);
865             return;
866           }
867           sibling_transformation(s);
868           return;
869         }
870         cleaning_transformation(s);
871       }
872       pair_transformation(q, s, heap_store);
873       assert(is_valid());
874     }
875
876     template <typename H>
877     void run_transformation(E* q, H& heap_store) {
878       assert((*q).type() == E::leader);
879       E* r = (*q).left();
880       assert((*r).type() == E::member);
881       E* p = (*q).parent();
882       if ((*p).right() == q) {
883         if (is_marked(p)) {
884           // std::cout << " only parent" << std::endl;
885           parent_transformation(q, heap_store);
886           return;
887         }
888         // std::cout << " first parent" << std::endl;
889         parent_transformation(q, heap_store);
890         if (! is_marked(q)) {
891           return;
892         }
893         if (is_marked((*p).left())) {
894           sibling_transformation((*p).left());
895           return;
896         }
897         // std::cout << " second parent" << std::endl;
898         parent_transformation(r, heap_store);
899         if (! is_marked(r)) {
900           return;
901         }
902         // std::cout << " third parent" << std::endl;
903         parent_transformation(r, heap_store);
904       }
905       else {
906         if (is_marked((*p).right())) {
907           sibling_transformation(q);
908           return;
909         }
910         // std::cout << " 1st cleaning " << std::endl;
911         cleaning_transformation(q);
912
913
914 #ifdef RANK
915         if (teams.size()>0) {
916           //  std::cout << "## special case " << std::endl;
917         return;
918         }
919 #endif
920         assert((*r).parent() == (*p).left());
921         if (is_marked((*r).parent() -> right())) {
922           sibling_transformation(r);
923           return;
924         }
925         /// std::cout << " 2nd cleaning " << std::endl;
```

```
926         cleanparent_transformation(r, heap_store);
927         //       cleaning_transformation(r);
928         // std::cout << " 1st parent " << std::endl;
929         //       parent_transformation(r, heap_store);
930         if (! is_marked(r)) {
931           return;
932         }
933         //       std::cout << " second sibling" << std::endl;
934         sibling_transformation(r);
935       }
936       assert(is_valid());
937     }
938
939   };
940 }
941
942 #if defined(UNITTEST_LAZY_MARK_STORE)
943 #include "proxy-list-heap-store.h++"
944 #include "fat-weak-heap-node.h++"
945 #include <functional>
946 #include <memory>
947 #include "heap-proxy.h++"
948 #include "multiple-heap-framework.h++"
949
950 template <typename V, typename E, typename A>
951 E* create(V const& v, A& allocator) {
952   E* p = allocator.allocate(1);
953   new (p) E(v, allocator);
954   return p;
955 }
956
957 template <typename V, typename E, typename A>
958 void destroy(E* p, A& allocator) {
959   p->~E();
960   allocator.deallocate(p, 1);
961 }
962
963 template <typename V, typename C, typename A, typename E,
964           typename H, typename M>
965 void test_mark_store() {
966   typedef cphstl::multiple_heap_framework<V, C, A, E, H, M> Q;
967   Q queue;
968
969   typedef typename A::template rebind<E>::other node_allocator_type;
970   node_allocator_type node_allocator;
971   E* p = create<V, E, node_allocator_type>(V(2), node_allocator);
972   E* q = create<V, E, node_allocator_type>(V(4), node_allocator);
973   E* r = create<V, E, node_allocator_type>(V(1), node_allocator);
974   E* s = create<V, E, node_allocator_type>(V(7), node_allocator);
975   E* t = create<V, E, node_allocator_type>(V(11), node_allocator);
976   E* u = create<V, E, node_allocator_type>(V(3), node_allocator);
977   E* v = create<V, E, node_allocator_type>(V(5), node_allocator);
978   E* w = create<V, E, node_allocator_type>(V(8), node_allocator);
979   E* x = create<V, E, node_allocator_type>(V(9), node_allocator);
980   E* y = create<V, E, node_allocator_type>(V(6), node_allocator);
981   E* z = create<V, E, node_allocator_type>(V(10), node_allocator);
982
983   queue.insert(p);
984   queue.insert(q);
985   queue.insert(r);
986   queue.insert(s);
987   queue.insert(t);
988   queue.insert(u);
989   queue.insert(v);
990   queue.insert(w);
```

```
991   queue.insert(x);
992   queue.insert(y);
993   queue.insert(z);
994   assert(queue.size() == 11);
995   assert(queue.top() == t);
996
997   H heap_store;
998   M mark_store;
999
1000  heap_store.inject(t, 3, mark_store);
1001  heap_store.inject(x, 1, mark_store);
1002  heap_store.inject(z, 0, mark_store);
1003
1004  assert(heap_store.is_valid(mark_store));
1005  assert(mark_store.is_valid());
1006
1007  mark_store.mark(u);
1008  assert(heap_store.is_valid(mark_store));
1009  assert(mark_store.is_valid());
1010  mark_store.mark(v);
1011  assert(heap_store.is_valid(mark_store));
1012  assert(mark_store.is_valid());
1013  mark_store.mark(w);
1014  assert(heap_store.is_valid(mark_store));
1015  assert(mark_store.is_valid());
1016  mark_store.mark(r);
1017  assert(heap_store.is_valid(mark_store));
1018  assert(mark_store.is_valid());
1019  mark_store.mark(s);
1020  assert(heap_store.is_valid(mark_store));
1021  assert(mark_store.is_valid());
1022  mark_store.mark(q);
1023  assert(heap_store.is_valid(mark_store));
1024  assert(mark_store.is_valid());
1025  mark_store.mark(p);
1026  assert(heap_store.is_valid(mark_store));
1027  assert(mark_store.is_valid());
1028  mark_store.mark(t);
1029  assert(heap_store.is_valid(mark_store));
1030  assert(mark_store.is_valid());
1031  mark_store.mark(x);
1032  assert(heap_store.is_valid(mark_store));
1033  assert(mark_store.is_valid());
1034  mark_store.mark(y);
1035  assert(heap_store.is_valid(mark_store));
1036  assert(mark_store.is_valid());
1037  mark_store.mark(z);
1038  assert(heap_store.is_valid(mark_store));
1039  assert(mark_store.is_valid());
1040
1041  for (int i = 0; i != 11; ++i) {
1042    mark_store.reduce(heap_store);
1043    assert(heap_store.is_valid(mark_store));
1044    assert(mark_store.is_valid());
1045  }
1046
1047  mark_store.unmark(z);
1048  assert(heap_store.is_valid(mark_store));
1049  assert(mark_store.is_valid());
1050  mark_store.unmark(y);
1051  assert(heap_store.is_valid(mark_store));
1052  assert(mark_store.is_valid());
1053  mark_store.unmark(x);
1054  assert(heap_store.is_valid(mark_store));
1055  assert(mark_store.is_valid());
```

```
1056   mark_store.unmark(t);
1057   assert(heap_store.is_valid(mark_store));
1058   assert(mark_store.is_valid());
1059   mark_store.unmark(p);
1060   assert(heap_store.is_valid(mark_store));
1061   assert(mark_store.is_valid());
1062   mark_store.unmark(q);
1063   assert(heap_store.is_valid(mark_store));
1064   assert(mark_store.is_valid());
1065   mark_store.unmark(s);
1066   assert(heap_store.is_valid(mark_store));
1067   assert(mark_store.is_valid());
1068   mark_store.unmark(r);
1069   assert(heap_store.is_valid(mark_store));
1070   assert(mark_store.is_valid());
1071   mark_store.unmark(w);
1072   assert(heap_store.is_valid(mark_store));
1073   assert(mark_store.is_valid());
1074   mark_store.unmark(v);
1075   assert(heap_store.is_valid(mark_store));
1076   assert(mark_store.is_valid());
1077   mark_store.unmark(u);
1078   assert(heap_store.is_valid(mark_store));
1079   assert(mark_store.is_valid());
1080
1081   std::pair<E*, std::size_t> small = heap_store.eject();
1082   assert(small.first == z);
1083   assert(small.second == 0);
1084
1085   std::pair<E*, std::size_t> medium = heap_store.eject();
1086   assert(medium.first == x);
1087   assert(medium.second == 1);
1088
1089   std::pair<E*, std::size_t> big = heap_store.eject();
1090   assert(big.first == t);
1091   assert(big.second == 3);
1092
1093   /* The above code destroyes owners.
1094   queue.show();
1095   while (queue.size() > 0) {
1096     (void) queue.extract();
1097   }
1098   */
1099
1100   destroy<V, E, node_allocator_type>(p, node_allocator);
1101   destroy<V, E, node_allocator_type>(q, node_allocator);
1102   destroy<V, E, node_allocator_type>(r, node_allocator);
1103   destroy<V, E, node_allocator_type>(s, node_allocator);
1104   destroy<V, E, node_allocator_type>(t, node_allocator);
1105   destroy<V, E, node_allocator_type>(u, node_allocator);
1106   destroy<V, E, node_allocator_type>(v, node_allocator);
1107   destroy<V, E, node_allocator_type>(w, node_allocator);
1108   destroy<V, E, node_allocator_type>(x, node_allocator);
1109   destroy<V, E, node_allocator_type>(y, node_allocator);
1110   destroy<V, E, node_allocator_type>(z, node_allocator);
1111 }
1112
1113 int main(int, char**) {
1114   typedef int V;
1115   typedef std::less<V> C;
1116   typedef std::allocator<V> A;
1117   typedef cphstl::fat_weak_heap_node<V, A> E;
1118   typedef cphstl::proxy_list_heap_store<C, A, E> H;
1119   typedef cphstl::lazy_mark_store<C, A, E> M;
1120
```

```
1121   test_mark_store<V, C, A, E, H, M>();
1122 }
1123
1124 #endif
1125 #endif
```

8.3 Priority-queue-frameworks/Code/eager-mark-store.h++

```
1 /*
2   An eager mark store
3
4   Author: Stefan Edelkamp © 2009
5 */
6
7 #ifndef __CPHSTL_EAGER_MARK_STORE__
8 #define __CPHSTL_EAGER_MARK_STORE__
9
10 #include "assert.h++"
11 #include "bit-store.h++"
12 #include "lazy-mark-store.h++"
13 #include <cstddef> // std::size_t
14 #include <cmath> // ilogb
15 #include <iostream>
16 #include <vector>
17
18 extern int ilogb(double) throw();
19
20 namespace cphstl {
21
22   template <typename C, typename A, typename E>
23   class eager_mark_store: public lazy_mark_store<C,A,E> {
24   public:
25
26     typedef C comparator_type;
27     typedef A allocator_type;
28     typedef E encapsulator_type;
29     typedef typename E::mark_type mark_type;
30     typedef typename E::height_type height_type;
31     typedef typename E::index_type index_type;
32     typedef unsigned long word_type;
33     typedef std::size_t size_type;
34
35     typedef eager_mark_store<C, A, E> M;
36     typedef lazy_mark_store<C, A, E> B;
37
38     struct entry {
39       encapsulator_type* position[3];
40       word_type matesize;
41     };
42
43   protected:
44
45     std::vector<entry*> entries;
46
47     B const* const up_cast(M const* const d) const {
48       return static_cast<B const* const>(d);
49     }
50
51     B* const up_cast(M* const d) const {
52       return static_cast<B* const>(d);
53     }
54
55   private:
56
57     eager_mark_store(eager_mark_store const&);
```

```
58      eager_mark_store& operator=(eager_mark_store const&);
59
60    public:
61
62      eager_mark_store(C const& c = C(), A const& a = A())
63        : lazy_mark_store<C,A,E>(c,a) {
64      }
65
66      ~eager_mark_store() {
67      }
68
69  #ifdef DEBUG
70
71      bool is_valid() {
72        return true;
73      }
74
75      void print() {
76        for(height_type h = 0; h < entries.size(); h++) {
77          entry* e1 = entries[h];
78          std::cout << " (" << h << ")"
79                    << e1->matesize
80                    << "[";
81          std::cout << e1->position[0] << "/";
82          if (e1->position[0])
83            std::cout << e1->position[0]->type();
84          std::cout << ",";
85
86          std::cout << e1->position[1] << "/";
87          if (e1->position[1])
88            std::cout << e1->position[1]->type();
89          std::cout << ",";
90
91          std::cout << e1->position[2] << "/";
92          if (e1->position[2])
93            std::cout << e1->position[2]->type();
94          std::cout << ",";
95        }
96        std::cout << " teams " << this->teams;
97        std::cout << " runs "  << this->runs;
98          std::cout << "\n";
99      }
100
101 #endif
102
103     size_type footprint(size_type n) const {
104       return
105         (ilogb(n) + 1) * (4 * sizeof(E*) +
106                           2 * sizeof(word_type)) +
107         2 * sizeof(word_type) +
108         2 * sizeof(word_type);
109     }
110
111     E* find_top() const {
112       if (this->runs || this->teams) {
113         E* top = 0;
114         for(height_type h = 0; h < entries.size(); h++) {
115           entry* e = entries[h];
116           if (top == 0) {
117             if (e->matesize > 1)
118               top = e->position[1];
119           }
120           else {
121             if(e->matesize == 2 &&
122                comparator(top->element(), e->position[1]->element()))
```

```
123             top = e->position[1];
124           }
125         }
126         return top;
127       }
128       else {
129         return 0;
130       }
131     }
132
133     template <typename H>
134     void reduce(H& heap_store) {
135       // std::cout << " reduce called "<< std::endl;
136       //  print();
137       assert(is_valid());
138       while (this->runs.size() || this->teams.size()) {
139         if (this->teams.size()) {
140            // std::cout << " case teams "<< std::endl;
141           word_type m = this->teams.choose();
142            // std::cout << " height " << m<< std::endl;
143           E* p = entries[m]->position[1];
144           assert((*p).type() == E::singleton);
145           E* q = entries[m]->position[2];
146           assert((*q).type() == E::singleton);
147
148            // std::cout << " before single transform "<< std::endl;
149           singleton_transformation(p, q, heap_store);
150            // std::cout << " after single transform "<< std::endl;
151         }
152         else { // (this->runs) {
153            // std::cout << " case runs "<< std::endl;
154           word_type m = this->runs.choose();
155           E* r = entries[m]->position[0];
156           //          entries[m]->position[0]->type() == E::leader ?
157           //          entries[m]->position[0] : entries[m]->position[1] ;
158           assert((*r).type() == E::leader);
159            // std::cout << " before run transform "<< std::endl;
160           run_transformation(r, heap_store);
161            // std::cout << " after run transform "<< std::endl;
162         }
163       }
164        // std::cout << " end reduce "<< std::endl;
165       //       print();
166
167 #ifndef NDEBUG
168
169       for(height_type h = 0; h < entries.size(); h++) {
170         entry* e1 = entries[h];
171
172         if (e1->position[0]) exit(1);
173         if (e1->position[1] && e1->position[2]) exit(1);
174       }
175
176 #endif
177
178     }
179
180     template <typename H>
181     void meld(eager_mark_store& other, H& heap_store) {
182
183       for (height_type h = 0; h < other.entries.size(); h++) {
184         entry* e1 = entries[h];
185         entry* e2 = other.entries[h];
186
187         assert(e1->matesize < 4 && e2->matesize < 4);
```

```
188
189         if (e2->matesize == 2) {
190           E* p = e2->position[1];
191           other.remove_mark(p);
192           other.remove_node(p);
193           insert_node(p);
194           insert_mark(p);
195           reduce(heap_store);
196         }
197       }
198
199       other.entries.resize(0); // "remove" all elements
200
201     }
202
203     void swap(eager_mark_store& other) {
204       // Precondition: The comparators are compatible.
205       std::swap(entries, other.entries);
206       std::swap(this->runs, other.runs);
207       // std::swap(singles, other.singles);
208       std::swap(this->teams, other.teams);
209     }
210
211   protected:
212
213     void insert_node(E* q) {
214       // entries.push_back(q);
215       // std::cout << "inserting node " << q << ": " << entries.size() << "/" << (*q
              ).height() + 1 << std::endl;
216
217       if (height_type((*q).height() + 1) > entries.size()) {
218         height_type max = std::max(entries.size(), size_type((*q).height() + 1));
219         // // std::cout << "expanding from depth " << entries.size() << " to depth "
                << max << std::endl;
220         height_type i=entries.size();
221         for (; i < max; i++) {
222           entry* e = new entry;
223           e->position[0] = e->position[1] = e->position[2] = 0;
224           e->matesize = 1;
225           entries.push_back(e) ;
226         }
227       }
228
229       height_type nheight = q->height();
230       entry* e = entries[nheight];
231
232       switch ((*q).type()) {
233
234       case E::member:
235         // std::cout << "newly insert member " << q << std::endl;
236         e->position[0] = q;
237         break;
238
239       case E::leader:
240         // std::cout << "node leader newly inserted " << std::endl;
241
242         if (nheight) {
243           entry* e1 = entries[nheight-1];
244           if (e1->position[0] == q->left())
245             break;
246           if (q->left()->type() == E::singleton) {
247             // std::cout << "left " << q->left() << " is singleton " << std::endl;
248             if(e1->position[1] == q->left())
249               e1->position[1] = e1->position[2];
250             e1->position[2] = 0;
```

```
251             e1->matesize--;
252             this->teams.unset(nheight-1);
253
254             e1->position[0] = q->left();
255             q->left()->type() = E::member;
256           }
257         }
258         break;
259
260       default:
261         ;
262       }
263     }
264
265     void insert_mark(E* q) {
266       // std::cout << "insert mark " << q << " type " << q->type() << std::endl;
267       // Precondition: (*q).type() must have the new value
268
269       height_type nheight = q->height();
270       entry* e = entries[nheight];
271       switch ((*q).type()) {
272
273       case E::leader:
274         // std::cout << "insert leader " << q << std::endl;
275
276         e->position[0] = q;
277         this->runs.set(nheight);
278         break;
279
280       case E::member:
281         // std::cout << "error member marked " << std::endl;
282         exit(1);
283         break;
284
285       case E::singleton:
286         // std::cout << "insert singleton " << q << std::endl;
287         e->position[0] = 0;
288
289 #ifndef NDEBUG
290         if (e->matesize > 3) {
291           // std::cout << "error mate insert "<< std::endl;
292           exit(1);
293         }
294 #endif
295
296         if (e->matesize > 1) // elect captain
297           this->teams.set(nheight);
298
299         e->position[e->matesize++] = q;
300         break;
301
302       default:
303         ;
304       }
305     }
306
307     void remove_mark(E* r) {
308       // Precondition: (*r).type() must have the old value
309        // std::cout << "removing mark " << r << std::endl;
310       height_type nheight = r->height();
311       entry* e = entries[nheight];
312
313       switch ((*r).type()) {
314       case E::leader:
315
```

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

9. Bit hacks

9.1 Priority-queue-frameworks/Code/bit-manipulation.h++

```
1 /*
2   Desc: Machine-dependant bit manipulation in terms of Intel
3   Hist: Imported from Vuillemin's priority queue
4   Warn: Cross-platform code
5   Authors: Asger Bruun, Jyrki Katajainen 2009, 2010
6 */
7
8 #ifndef __CPHSTL_BIT_MANIPULATION__
9 #define __CPHSTL_BIT_MANIPULATION__
10
11 #include "assert.h++"
12 #include <climits> // CHAR_BIT
13 #include <cstddef>
14
15 namespace cphstl {
16
17   template<typename T>
18   std::size_t leading_zeros(T n) {
19     assert(n != 0);
20     T b(0);
21     while (n != 0) {
22       n >>= 1;
23       ++b;
24     }
25     return sizeof(T) * CHAR_BIT - b;
26   }
27
28   template<typename T>
29   std::size_t trailing_zeros(T n) {
30     assert(n != 0);
31     T b(0);
32     while ((n & 1) == 0) {
33       n >>= 1;
34       ++b;
35     }
36     return b;
37   }
38
39   template<typename T>
40   std::size_t population_count(T n) {
41     assert(n != 0);
42     T b(0);
43     while (n != 0) {
44       b += (n & 1);
45       n >>= 1;
46     }
47     return b;
48   }
49 }
50
51 #ifdef _MSC_VER
52 #include <intrin.h> // _BitScanForward
53 #include <cstdlib>
54
55 namespace cphstl {
56
57 #if defined(_M_X86)
58 #pragma intrinsic(_bittest, _BitScanForward,_BitScanReverse,__popcnt)
59
60   bool bit_test(long const& n, long pos) {
```

```
61      return bool(_bittest((long *) &n, pos));
62    }
63
64    std::size_t leading_zeros(unsigned long n) {
65      assert(n != 0);
66      unsigned long index;
67      (void) _BitScanForward(&index, n);
68      return sizeof(std::size_t) * CHAR_BIT - index - 1;
69    };
70
71    std::size_t trailing_zeros(unsigned long n) {
72      assert(n != 0);
73      unsigned long index;
74      (void) _BitScanReverse(&index, n);
75      return index;
76    };
77
78  #elif defined(_M_X64)
79  #pragma intrinsic(_BitScanForward,_BitScanReverse,__popcnt)
80
81    bool bit_test(std::size_t const& n, std:.size_t pos) {
82      return bool(_bittest64((std::__int64 *) &n, pos));
83    }
84
85    std::size_t leading_zeros(unsigned std::__int64 n) {
86      assert(n != 0);
87      unsigned long index;
88      (void) _BitScanForward(&index, n);
89      return sizeof(std::size_t) * CHAR_BIT - index - 1;
90    };
91
92    std::size_t trailing_zeros(unsigned std::__int64 n) {
93      assert(n != 0);
94      unsigned long index;
95      (void) _BitScanReverse64(&index, n);
96      return index;
97    };
98
99    std::size_t inline population_count(unsigned std::__int64 n) {
100     assert(n != 0);
101     return __popcnt64(n);
102   };
103
104 #endif
105
106 }
107
108 #elif defined(__GNUC__)
109
110 namespace cphstl {
111   // Ref: gcc.gnu.org/onlinedocs/gcc-4.4.0/gcc/Other-Builtins.html#index-
          g_t_005f_005fbuiltin_005fffs-2894
112
113   std::size_t leading_zeros(unsigned long long n) {
114     assert(n != 0);
115     return __builtin_clzll(n);
116   }
117
118   std::size_t trailing_zeros(unsigned long long n) {
119     assert(n != 0);
120     return __builtin_ctzll(n);
121   }
122
123   std::size_t population_count(unsigned long long n) {
124     return __builtin_popcountll(n);
```

```
125   }
126
127   std::size_t leading_zeros(unsigned long n) {
128     assert(n != 0);
129     return __builtin_clzl(n);
130   }
131
132   std::size_t trailing_zeros(unsigned long n) {
133     assert(n != 0);
134     return __builtin_ctzl(n);
135   }
136
137   std::size_t population_count(unsigned long n) {
138     return __builtin_popcountl(n);
139   }
140
141   std::size_t leading_zeros(unsigned int n) {
142     assert(n != 0);
143     return __builtin_clz(n);
144   }
145
146   std::size_t trailing_zeros(unsigned int n) {
147     assert(n != 0);
148     return __builtin_ctz(n);
149   }
150
151   std::size_t population_count(unsigned int n) {
152     return __builtin_popcount(n);
153   }
154 }
155
156 #endif
157 #endif
```

9.2 Priority-queue-frameworks/Code/bit-store.h++

```
1  /*
2    A bit store keeps a sequence of bits in a single word. Requirement:
3    The length of the sequence should not be larger than the size of a
4    word measured in bits.
5
6    Author: Jyrki Katajainen © 2009, 2010
7  */
8
9  #ifndef __CPHSTL_BIT_STORE____
10 #define __CPHSTL_BIT_STORE____
11
12 #include "assert.h++"
13 #include "bit-manipulation.h++"
14 #include <climits> // CHAR_BIT
15 #include <cstddef>
16 #include <iostream>
17
18 namespace cphstl {
19
20   template <typename W>
21   class bit_store;
22
23   template <>
24   class bit_store<unsigned long> {
25   public:
26
27     typedef bool value_type;
28     typedef unsigned long word_type;
29     typedef std::size_t size_type;
```

```
30
31    enum {word_size = CHAR_BIT * sizeof(word_type)};
32
33    explicit bit_store(word_type value = 0)
34      : word(value) {
35    }
36
37    operator word_type() const {
38      return word;
39    }
40
41    size_type size() const {
42      return population_count(word);
43    }
44
45    size_type capacity() const {
46      return word_size;
47    };
48
49    template <typename I>
50    void set(I index) {
51      assert(word_type(index) < word_size);
52      word_type mask = word_type(1) << word_type(index);
53      word &= ~ mask;
54      word |= mask;
55    }
56
57    template <typename I>
58    void unset(I index) {
59      assert(word_type(index) < word_size);
60      word_type mask = word_type(1) << word_type(index);
61      word &= ~ mask;
62    }
63
64    template <typename I>
65    bool get(I index) const {
66      assert(index < word_size);
67      word_type v = word_type(1) << index;
68      v = word & v;
69      return (v > 0);
70    }
71
72    size_type least_significant_one() const {
73      assert(size() != 0);
74      return trailing_zeros(word);
75    }
76
77    size_type most_significant_one() const {
78      assert(size() != 0);
79      return capacity() - leading_zeros(word) - 1;
80    }
81
82    size_type choose() const {
83      assert(size() != 0);
84      return most_significant_one();
85    }
86
87  protected:
88
89    word_type word;
90  };
91
92  template <>
93  class bit_store<unsigned long long> {
94  public:
```

```
95
96      typedef bool value_type;
97      typedef unsigned long long word_type;
98      typedef std::size_t size_type;
99
100     enum {word_size = CHAR_BIT * sizeof(word_type)};
101
102     explicit bit_store(word_type value = 0)
103       : word(value) {
104     }
105
106     operator word_type() const {
107       return word;
108     }
109
110     size_type size() const {
111       return population_count(word);
112     }
113
114     size_type capacity() const {
115       return word_size;
116     };
117
118     template <typename I>
119     void set(I index) {
120       assert(word_type(index) < word_size);
121       word_type mask = word_type(1) << word_type(index);
122       word &= ~ mask;
123       word |= mask;
124     }
125
126     template <typename I>
127     void unset(I index) {
128       assert(word_type(index) < word_size);
129       word_type mask = word_type(1) << word_type(index);
130       word &= ~ mask;
131     }
132
133     template <typename I>
134     bool get(I index) const {
135       assert(index < word_size);
136       word_type v = word_type(1) << index;
137       v = word & v;
138       return (v > 0);
139     }
140
141     int least_significant_one() const {
142       assert(size() != 0);
143       return trailing_zeros(word);
144     }
145
146     int most_significant_one() const {
147       assert(size() != 0);
148       return capacity() - leading_zeros(word) - 1;
149     }
150
151     int choose() const {
152       assert(size() != 0);
153       return most_significant_one();
154     }
155
156   protected:
157
158     word_type word;
159   };
```

```
160 }
161
162 #if defined(UNITTEST_BIT_STORE)
163 #include "assert.h++"
164 #include <iostream>
165
166 template <typename T>
167 class unittest_bit_store {
168 public:
169
170   void operator()() {
171     cphstl::bit_store<T> word;
172     T n = word; // convertion operator
173     assert(n == 0);
174     T capacity = word.capacity(); // capacity
175     assert(capacity == sizeof(T) * CHAR_BIT);
176     word.set(0);
177     word.set(0);
178     assert(word == 1); // set
179     bool bit_zero = word.get(0);
180     assert(bit_zero == 1); // get
181     bool bit_one = word.get(1);
182     assert(bit_one == 0);
183     bool bit_two = word.get(2);
184     assert(bit_two == 0);
185     word.set(1);
186     assert(word == 3);
187     assert(word.size() == 2); // size
188     std::size_t i = word.choose();
189     assert(i == 1);
190     word.set(1);
191     assert(word.size() == 2);
192     word.unset(1); // unset
193     word.unset(1); // unset
194     assert(word.size() == 1);
195     i = word.choose();
196     assert(i == 0);
197   }
198 };
199
200 int main(int, char**) {
201   unittest_bit_store<unsigned long> s;
202   s();
203   unittest_bit_store<unsigned long long> t;
204   t();
205   return 0;
206 }
207
208 #endif
209 #endif
```

10. Iterator

10.1 Iterator/Code/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 <cstddef> // std::ptrdiff_t
19  #include <iostream> // std::ostream
20  #include <iterator> // std::forward_iterator_tag
21  #include <string> // std::string
22
23  namespace {
24
25    // if statement for compile-time meta-programming
26
27    template <bool, typename T, typename U>
28    class if_then_else;
29
30    template <typename T, typename U>
31    class if_then_else<true, T, U> {
32    public:
33      typedef T type;
34    };
35
36    template <typename T, typename U>
37    class if_then_else<false, T, U> {
38    public:
39      typedef U type;
40    };
41  }
42
43  namespace cphstl {
44
45    template <typename V, typename C, typename A, typename F,
46              typename R, typename I, typename J>
47    class meldable_priority_queue;
48
49    template <typename E, typename R, bool is_const = false>
50    class priority_queue_iterator {
51    public:
52
53      // types
54
55      typedef std::forward_iterator_tag iterator_category;
56      typedef E encapsulator_type;
57      typedef R realizator_type;
58      typedef typename E::value_type value_type;
59      typedef std::ptrdiff_t difference_type;
60
61      typedef typename if_then_else<is_const, value_type const*, value_type*>::type
          pointer;
62      typedef typename if_then_else<is_const, value_type const&, value_type&>::type
          reference;
63
64      typedef priority_queue_iterator<E, R, !is_const> complement;
65
66    private:
67
68      typedef typename if_then_else<is_const, E const*, E*>::type node_pointer;
69
70    public:
71
72      // friends
```

```
73
74     friend class priority_queue_iterator<E, R, !is_const>;
75
76     template <typename F, typename Q, bool both>
77     friend std::ostream& operator<<(std::ostream&, priority_queue_iterator<F, Q,
           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     priority_queue_iterator(priority_queue_iterator<E, R, false> const&);
87     priority_queue_iterator& operator=(priority_queue_iterator const&);
88     ~priority_queue_iterator();
89
90     // operators
91
92     reference operator*() const;
93     pointer operator->() const;
94     pointer operator->();
95     priority_queue_iterator& operator++();
96     priority_queue_iterator operator++(int);
97     priority_queue_iterator& operator--();
98     priority_queue_iterator operator--(int);
99
100    template <bool both>
101    bool operator==(priority_queue_iterator<E, R, both> const&) const;
102
103    template <bool both>
104    bool operator!=(priority_queue_iterator<E, R, both> const&) const;
105
106  private:
107
108    // converters to be used by the friends
109
110    priority_queue_iterator(node_pointer, R*);
111    operator node_pointer() const;
112    operator std::string() const;
113
114    node_pointer link;
115  };
116 }
117
118 #include "priority-queue-iterator.i++"
119 #endif
```

10.2 Iterator/Code/priority-queue-iterator.i++

```
1 /*
2   Implementation of cphstl::priority_queue_iterator
3
4   Author: Jyrki Katajainen © 2009, 2010
5 */
6
7 #include "assert.h++"
8 #include <sstream> // defines std::stringstream
9
10 namespace cphstl {
11
12   // default constructor
13
14   template <typename E, typename R, bool is_const>
```

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

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

```
141   std::ostream&
142   operator<<(std::ostream& s, priority_queue_iterator<E, R, both> const& i) {
143     s << std::string(i);
144     return s;
145   }
146
147 }
```

10.3 Iterator/Code/proxy-iterator.h++

```
1 /*
2   The proxy iterator stores a pointer to an encapsulator and a pointer
3   to a surrogate. The pointer to the encapsulator represents the
4   current position from where the value can be fetched. The surrogate
5   points to a kernel. The surrogate is used to advance the iterator,
6   where the access member function is called to get a pointer to the
7   desired encapsulator.
8
9   The idea of combining iterators and const iterators into the same
10  class is taken from [Matt Austern. Defining iterators and const
11  iterators. C/C++ User's Journal 19,1 (2001), 74-79].
12
13  Authors: Jyrki Katajainen, Bo Simonsen © 2008, 2010
14 */
15
16 #ifndef __CPHSTL_PROXY_ITERATOR__
17 #define __CPHSTL_PROXY_ITERATOR__
18
19 #include <cstddef> // std::size_t and std::ptrdiff_t
20 #include <iterator> // std::random_access_iterator_tag
21 #include "type.h++" // cphstl::if_then_else
22 #include <utility> // std::pair
23
24 namespace cphstl {
25
26   template <typename V, typename A, typename K, typename I, typename J>
27   class vector;
28
29   template <typename V, typename C, typename A, typename F, typename R,
30             typename I, typename J>
31   class meldable_priority_queue;
32
33   template <typename E, typename R, bool is_const = false>
34   class proxy_iterator {
35   public:
36
37     // types
38
39     typedef E encapsulator_type;
40     typedef R realizator_type;
41     typedef std::random_access_iterator_tag iterator_category;
42     typedef typename E::value_type value_type;
43     typedef std::size_t size_type;
44     typedef std::ptrdiff_t difference_type;
45     typedef typename if_then_else<is_const, value_type const*, value_type*>::type
            pointer;
46     typedef typename if_then_else<is_const, value_type const&, value_type&>::type
            reference;
47
48   protected:
49
50     // types
51
52     typedef typename R::surrogate_type surrogate_type;
53     typedef typename if_then_else<is_const, E const*, E*>::type node_pointer;
```

```
54
55  public:
56
57    // friends
58
59    friend class proxy_iterator<E, R, !is_const>;
60
61    template <typename V, typename A, typename K, typename I, typename J>
62    friend class cphstl::vector;
63
64    template <typename V, typename C, typename A, typename F, typename S,
65              typename I, typename J>
66    friend class cphstl::meldable_priority_queue;
67
68    // structors
69
70    proxy_iterator();
71    proxy_iterator(proxy_iterator<E, R, false> const&);
72    proxy_iterator(proxy_iterator<E, R, true> const&);
73    proxy_iterator& operator=(proxy_iterator const&);
74    ~proxy_iterator();
75
76    // operators
77
78    reference operator*() const;
79    pointer operator->() const;
80    proxy_iterator& operator++();
81    proxy_iterator operator++(int);
82    proxy_iterator& operator--();
83    proxy_iterator operator--(int);
84    proxy_iterator& operator+=(difference_type);
85    proxy_iterator& operator-=(difference_type);
86    proxy_iterator operator+(difference_type) const;
87    proxy_iterator operator-(difference_type) const;
88    difference_type operator-(proxy_iterator const&) const;
89
90    template <bool both>
91    bool operator==(proxy_iterator<E, R, both> const&) const;
92
93    template <bool both>
94    bool operator!=(proxy_iterator<E, R, both> const&) const;
95
96    template <bool both>
97    bool operator<(proxy_iterator<E, R, both> const&) const;
98
99    template <bool both>
100   bool operator>(proxy_iterator<E, R, both> const&) const;
101
102   template <bool both>
103   bool operator<=(proxy_iterator<E, R, both> const&) const;
104
105   template <bool both>
106   bool operator>=(proxy_iterator<E, R, both> const&) const;
107
108 protected:
109
110   // converters to be used by the container friends
111
112   proxy_iterator(node_pointer, R*);
113   operator node_pointer() const;
114
115   void advance(difference_type n);
116
117   node_pointer position;
118   surrogate_type* surrogate;
```

```
119   };
120
121   template<typename E, typename R, bool both>
122   proxy_iterator<E, R, both>
123   operator+(std::ptrdiff_t, proxy_iterator<E, R, both> const&);
124
125 }
126
127 #include "proxy-iterator.i++" // implements cphstl::proxy_iterator
128
129 #endif
```

10.4 Iterator/Code/proxy-iterator.i++

```
1  /*
2    Implementation of cphstl::proxy_iterator
3
4    Authors: Jyrki Katajainen, Bo Simonsen © 2008, 2010
5  */
6
7  #include "assert.h++"
8  #include <cstdlib>
9  #include <iostream>
10
11 namespace cphstl {
12
13   // default constructor
14
15   template <typename E, typename R, bool is_const>
16   proxy_iterator<E, R, is_const>::proxy_iterator()
17     : position(node_pointer()), surrogate(0) {
18   }
19
20   // copy constructor
21
22   template <typename E, typename R, bool is_const>
23   proxy_iterator<E, R, is_const>::proxy_iterator(proxy_iterator<E, R, false> const&
         a)
24     : position(a.position), surrogate(a.surrogate) {
25   }
26
27    template <typename E, typename R, bool is_const>
28    proxy_iterator<E, R, is_const>::proxy_iterator(proxy_iterator<E, R, true> const&
          a)
29      : position(a.position), surrogate(a.surrogate) {
30   }
31
32   // assignment
33
34   template <typename E, typename R, bool is_const>
35   proxy_iterator<E, R, is_const>&
36   proxy_iterator<E, R, is_const>::operator=(proxy_iterator<E, R, is_const> const& a)
          {
37     position = a.position;
38     surrogate = a.surrogate;
39     return *this;
40   }
41
42   // destructor
43
44   template <typename E, typename R, bool is_const>
45   proxy_iterator<E, R, is_const>::~proxy_iterator() {
46   }
47
48   // operator*
```

```
49
50  template <typename E, typename R, bool is_const>
51  typename proxy_iterator<E, R, is_const>::reference
52  proxy_iterator<E, R, is_const>::operator*() const {
53    return reference((*position).element());
54  }
55
56  // operator->
57
58  template <typename E, typename R, bool is_const>
59  typename proxy_iterator<E, R, is_const>::pointer
60  proxy_iterator<E, R, is_const>::operator->() const {
61    return pointer(&(*position).element());
62  }
63
64  template <typename E, typename R, bool is_const>
65  void
66  proxy_iterator<E, R, is_const>::advance(difference_type n) {
67    if (n == 0) {
68      return;
69    }
70    surrogate_type s = *surrogate;
71    if (position == 0) {
72      difference_type i = s.subject()->size() + n;
73      position = s.access(i);
74    }
75    else {
76      difference_type i = std::abs(difference_type((*position).position())) + n;
77      if (i >= difference_type(s.subject()->size()) || i < 0) {
78        position = 0;
79      }
80      else {
81        position = s.access(i);
82      }
83    }
84  }
85
86  // operator++; pre-increment
87
88  template <typename E, typename R, bool is_const>
89  proxy_iterator<E, R, is_const>&
90  proxy_iterator<E, R, is_const>::operator++() {
91    advance(1);
92    return *this;
93  }
94
95  // operator++; post-increment
96
97  template <typename E, typename R, bool is_const>
98  proxy_iterator<E, R, is_const>
99  proxy_iterator<E, R, is_const>::operator++(int) {
100   proxy_iterator<E, R, is_const> temporary = *this;
101   advance(1);
102   return temporary;
103 }
104
105 // operator--; pre-decrement
106
107 template <typename E, typename R, bool is_const>
108 proxy_iterator<E, R, is_const>&
109 proxy_iterator<E, R, is_const>::operator--() {
110   advance(-1);
111   return *this;
112 }
113
```

```
114   // operator--; post-decrement
115
116   template <typename E, typename R, bool is_const>
117   proxy_iterator<E, R, is_const>
118   proxy_iterator<E, R, is_const>::operator--(int) {
119     proxy_iterator<E, R, is_const> temporary = *this;
120     advance(-1);
121     return temporary;
122   }
123
124   // operator+=
125
126   template <typename E, typename R, bool is_const>
127   proxy_iterator<E, R, is_const>&
128   proxy_iterator<E, R, is_const>::operator+=(difference_type n) {
129     advance(n);
130     return *this;
131   }
132
133   // operator-=
134
135   template <typename E, typename R, bool is_const>
136   proxy_iterator<E, R, is_const>&
137   proxy_iterator<E, R, is_const>::operator-=(difference_type n) {
138     advance(-n);
139     return *this;
140   }
141
142   // operator+
143
144   template <typename E, typename R, bool is_const>
145   proxy_iterator<E, R, is_const>
146   proxy_iterator<E, R, is_const>::operator+(difference_type n) const {
147     proxy_iterator<E, R, is_const> temporary = *this;
148     temporary.advance(n);
149     return temporary;
150   }
151
152   // operator-
153
154   template <typename E, typename R, bool is_const>
155   proxy_iterator<E, R, is_const>
156   proxy_iterator<E, R, is_const>::operator-(difference_type n) const {
157     proxy_iterator<E, R, is_const> temporary = *this;
158     temporary.advance(-n);
159     return temporary;
160   }
161
162   // iterator distance
163
164   template <typename E, typename R, bool is_const>
165   typename proxy_iterator<E, R, is_const>::difference_type
166   proxy_iterator<E, R, is_const>::operator-(proxy_iterator<E, R, is_const> const& a)
          const {
167     if (position == 0 && a.position == 0) {
168       return 0;
169     }
170     else if (position == 0) {
171       difference_type n = (*(*surrogate).subject()).size();
172       difference_type i = std::abs((*a.position).position());
173       return n - i;
174     }
175     else if (a.position == 0) {
176       difference_type i = std::abs((*position).position());
177       difference_type n = (*(*surrogate).subject()).size();
```

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

```
237   proxy_iterator<E, R, is_const>::operator>=(proxy_iterator<E, R, both> const& a)
          const {
238     return !(*this < a);
239   }
240
241   // parametrized constructors
242
243   template <typename E, typename R, bool is_const>
244   proxy_iterator<E, R, is_const>::proxy_iterator(node_pointer p, realizator_type* r)
245     : position(p), surrogate((*r).surrogate) {
246   }
247
248   // convertion operator
249
250   template <typename E, typename R, bool is_const>
251   proxy_iterator<E, R, is_const>::operator node_pointer() const {
252     return position;
253   }
254
255   // operator+(int, iterator)
256
257   template <typename E, typename R, bool is_const>
258   proxy_iterator<E, R, is_const> operator+(std::ptrdiff_t n, proxy_iterator<E, R,
          is_const> const& a) {
259     return a + n;
260   }
261
262 }
```

11. Proxies

11.1 Proxy/Code/allocator-proxy.h++

```
1 #ifndef __CPHSTL_ALLOCATOR_PROXY__
2 #define __CPHSTL_ALLOCATOR_PROXY__
3
4 /*
5   A generic allocator proxy class. It is actually more a real
6   encapsulator class as described in [1]. It keeps an instance
7   of the allocator, given by the parameterized constructor,
8   as a pointer. The allocator instance is allocated. The purpose of
9   this class is to perform a safe swap operation for all containers.
10  Swap must not throw an exception, so when swapping this proxy class
11  we swap pointers which cannot fail.
12  All STL containers keeps an allocator which makes this class generic.
13
14  [1] Pascoe, Geoffrey A. Encapsulators: a new software paradigm in
15      Smalltalk-80. SIGPLAN Notices 21(11), 1986, 341-346.
16
17  Author: Bo Simonsen © 2009
18 */
19
20 #include <algorithm>
21
22 namespace cphstl {
23   template <typename A>
24   class allocator_proxy {
25   public:
26     typedef typename A::size_type size_type;
27     typedef typename A::difference_type difference_type;
28     typedef typename A::pointer pointer;
29     typedef typename A::const_pointer const_pointer;
30     typedef typename A::reference reference;
31     typedef typename A::const_reference const_reference;
```

```
32      typedef typename A::value_type value_type;
33
34      template <class U>
35      struct rebind { typedef allocator_proxy<typename A::template rebind<U>::other >
            other; };
36
37      allocator_proxy() {
38        (*this).a = new A();
39      }
40      allocator_proxy(A const& a) {
41        (*this).a = new A(a);
42      }
43      ~allocator_proxy() {
44        delete (*this).a;
45      }
46      allocator_proxy(allocator_proxy const& ap) {
47        (*this).a = new A(*ap.a);
48      }
49      allocator_proxy(allocator_proxy& ap) {
50        (*this).a = new A(*ap.a);
51      }
52      allocator_proxy operator=(allocator_proxy const& ap) {
53        delete (*this).a;
54        (*this).a = new A(*ap.a);
55        return (*this);
56      }
57      allocator_proxy operator=(A const& a) {
58        delete (*this).a;
59        (*this).a = new A(a);
60        return (*this);
61      }
62      pointer address(reference ref) const {
63        return (*a).address(ref);
64      }
65      const_pointer address(const_reference const_ref) const {
66        return (*a).address(const_ref);
67      }
68      size_type max_size() const throw() {
69        return (*a).max_size();
70      }
71      pointer allocate(size_type s, const_pointer p = 0) {
72        return (*a).allocate(s);
73      }
74      void construct(pointer p,value_type const& v) {
75        (*a).construct(p,v);
76      }
77      void destroy(pointer p){
78        (*a).construct(p);
79      }
80      void deallocate(pointer p,size_type s) {
81        (*a).deallocate(p,s);
82      }
83      A& subject() const {
84        return *a;
85      }
86    private:
87      A* a;
88
89    };
90
91  }
92
93  #endif
```

11.2 Proxy/Code/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 #include <iostream>
11
12 namespace cphstl {
13
14   template <typename C>
15   class comparator_proxy {
16   public:
17
18     comparator_proxy(C const& c = C()) {
19       (*this).c = new C(c);
20     }
21
22     comparator_proxy(comparator_proxy const& cp) {
23       (*this).c = new C(*cp.c);
24     }
25
26     comparator_proxy operator=(comparator_proxy const& cp) {
27       delete (*this).c;
28       (*this).c = new C(*cp.c);
29       return (*this);
30     }
31
32     comparator_proxy operator=(C const& c) {
33       delete (*this).c;
34       (*this).c = new C(c);
35       return (*this);
36     }
37
38     ~comparator_proxy() {
39       delete (*this).c;
40     }
41
42     template <typename T, typename U>
43     bool operator()(T const& t, U const& u) const {
44       return (*c).operator()(t, u);
45     }
46
47     C subject() const {
48       return *c;
49     }
50
51   private:
52
53     C* c;
54   };
55 }
56
57 #endif
```

11.3 Proxy/Code/vector-surrogate.h++

```
1 #ifndef __CPHSTL_VECTOR_SURROGATE__
2 #define __CPHSTL_VECTOR_SURROGATE__
```

```
3
4 /*
5   This surrogate keeps a pointer to a vector and accesses its elements
6   on behalf of it.
7
8   Author: Jyrki Katajainen © 2010
9 */
10
11 namespace cphstl {
12
13   template <typename S>
14   class vector_surrogate {
15   public:
16
17     typedef S subject_type;
18     typedef typename S::value_type return_type;
19
20     subject_type*& subject() {
21       return ptr;
22     }
23
24     template <typename I>
25     return_type access(I index) {
26       return (*ptr)[index];
27     }
28
29   private:
30
31     subject_type* ptr;
32   };
33 }
34
35 #endif
```

12. Unit tests

12.1 Assert/Code/assert.h++

```
1 /*
2   The assert macro is taken quite directly from the book [Steve
3   Maguire, Writing solid code, Microsoft Press (1993)]. This macro is
4   a statement that cannot be used in expression context.
5
6   The static assert is now provided by the compiler.
7
8   Author: Jyrki Katajainen © 2001, 2008
9 */
10
11 #ifndef __CPHSTL_ASSERT__
12 #define __CPHSTL_ASSERT__
13
14 #include <cstdio> // std::fflush, std::fprintf
15 #include <cstdlib> // std::abort
16
17 namespace cphstl {
18
19 #ifdef NDEBUG
20 #define assert(condition)
21 #else
22
23   void __assert(char const* message, char const* strFile, unsigned int uLine) {
24     std::fflush(NULL);
25     std::fprintf(stderr, "\nAssertion '%s' failed: %s, line %u\n",
26                  message, strFile, uLine);
```

```
27     std::fflush(stderr);
28     std::abort();
29   }
30
31 #define assert(condition) \
32   if (condition) { \
33   } \
34   else \
35     cphstl::__assert(#condition, __FILE__, __LINE__)
36 #endif
37
38 }
39
40 #endif
```

12.2 Meldable-priority-queue/Test/smoke-test.c++

```
 1 /*
 2   Smoke test for meldable priority queues
 3
 4   Authors: Claus Jensen, Jyrki Katajainen © 2006, 2009, 2010
 5 */
 6
 7 #include <algorithm> // std::copy, std::equal
 8 #include "assert.h++"
 9 #include <cstdlib> // std::size_t
10 #include "eager-mark-store.h++"
11 #include "element-encapsulator.h++"
12 #include "fat-weak-heap-node.h++"
13 #include <functional> // std::less
14 #include "input_generation.c++" // data generation routines
15 #include <iostream>
16 #include "lazy-mark-store.h++"
17 #include "pennant-node.h++"
18 #include "multiple-heap-framework.h++"
19 #include "priority-queue-iterator.h++"
20 #include "proxy-iterator.h++"
21 #include "proxy-array-heap-store.h++"
22 #include "proxy-list-heap-store.h++"
23 #include "simple-mark-store.h++"
24 #include "single-heap-framework.h++"
25 #include "stl-meldable-priority-queue.h++"
26 #include "top-down-binary-heap-heapifier.h++"
27 #include "vector-surrogate.h++"
28 #include "weak-heap-heapifier.h++"
29 #include "weak-heap-node.h++"
30
31 template <typename T>
32 void ignore_unused_variable_warning(T const&) {
33 }
34
35 using namespace cphstl;
36
37 template <typename V, typename C, typename A, typename N, typename R,
38           typename I, typename J>
39 void test_members() {
40   typedef meldable_priority_queue<V, C, A, N, R, I, J> Q;
41
42   V in;
43
44   typename Q::value_type* p_val = (V*) 0;
45   ignore_unused_variable_warning(p_val);
46   typename Q::comparator_type* p_comp = (C*) 0;
47   ignore_unused_variable_warning(p_comp);
48   typename Q::allocator_type* p_alloc = (A*) 0;
```

```
49  ignore_unused_variable_warning(p_alloc);
50  typename Q::pointer p_ptr = (V*) 0;
51  ignore_unused_variable_warning(p_ptr);
52  typename Q::const_pointer cp_ptr = (V const*) 0;
53  ignore_unused_variable_warning(cp_ptr);
54  typename Q::reference p_ref = in;
55  ignore_unused_variable_warning(p_ref);
56  typename Q::const_reference cp_ref = (V const&) in;
57  ignore_unused_variable_warning(cp_ref);
58  typename Q::size_type* p_size = (std::size_t*) 0;
59  ignore_unused_variable_warning(p_size);
60  typename Q::difference_type* p_diff = (std::ptrdiff_t*) 0;
61  ignore_unused_variable_warning(p_diff);
62
63  Q q0;
64  A al = q0.get_allocator();
65  C comp = q0.get_comparator();
66  Q q0a(comp), q0b(comp, al);
67  assert(q0.empty() && q0.size() == 0);
68  assert(q0.size() <= q0.max_size());
69  assert(q0a.size() == 0 && q0a.get_allocator() == al);
70  assert(q0b.size() == 0 && q0b.get_allocator() == al);
71  Q q1;
72  q1.push(1);
73  q1.push(2);
74  q1.push(3);
75  assert(q1.size() == 3 && *q1.top() == 3);
76
77  Q q2(comp);
78  q2.push(1);
79  q2.push(2);
80  q2.push(3);
81  assert(q2.size() == 3 && *q2.top() == 3);
82  Q q3(comp, al);
83  q3.push(1);
84  q3.push(2);
85  q3.push(3);
86  assert(q3.size() == 3 && *q3.top() == 3);
87
88  Q q4;
89  q4.push(1);
90  q4.push(2);
91  q4.push(3);
92  q0 = q4;
93  q0.swap(q4);
94  assert(q0.size() == 3 && *q0.top() == 3);
95  q1.meld(q2);
96  assert(q1.size() == 6 && *q1.top() == 3);
97
98  typename Q::iterator p_it(q0.begin());
99  typename Q::const_iterator p_cit(q0.begin());
100
101 assert(*p_it == *p_cit);
102 q0.pop();
103 q0.pop();
104 p_it = q0.begin();
105 assert(*p_it == *q0.top());
106 p_cit = q0.begin();
107 assert(*p_cit == *q0.top());
108
109 q0.clear();
110 typename Q::iterator p_it4 = q0.push(4);
111 assert(*p_it4 == 4);
112 typename Q::iterator p_it5 = q0.push(5);
113 assert(*q0.top() == 5);
```

```
114   V const c_value = 6;
115   q0.increase(p_it4, c_value);
116   assert(*q0.top() == 6);
117   q0.erase(p_it4);
118   assert(*q0.top() == 5);
119   typename Q::iterator p_it3 = q0.push(3);
120   q0.pop();
121   assert(*q0.top() == 3);
122
123   q0.clear();
124   assert(q0.empty());
125   q0.swap(q3);
126   assert(!q0.empty() && q3.empty());
127   q3.swap(q0);
128   assert(q0.empty() && !q3.empty());
129   q0.clear();
130   q0a.clear();
131   q0b.clear();
132   q1.clear();
133   q2.clear();
134   q3.clear();
135   q4.clear();
136 }
137
138 template <typename V, typename C, typename A, typename N, typename R,
139           typename I, typename J>
140 void test_queue_sort() {
141   typedef meldable_priority_queue<V, C, A, N, R, I, J> Q;
142
143   unsigned int n = 100;
144   V a[100];
145   random_sequence(a, a + n);
146   V b[100];
147   std::copy(a, a + n, b);
148   Q q5;
149   for (unsigned int k = 0; k < n; ++k) {
150     q5.push(a[k]);
151   }
152   for (unsigned int k = 0; k < n; ++k) {
153     a[n - 1 - k] = *q5.top();
154     q5.pop();
155   }
156   std::sort(b, b + n, C());
157   assert(std::equal(a, a + n, b));
158   assert(q5.size() == 0);
159 }
160
161 int main() {
162   typedef int V;
163   typedef std::less<V> C;
164   typedef std::allocator<V> A;
165
166   typedef cphstl::element_encapsulator<V, std::size_t, A> N0;
167   typedef cphstl::single_heap_framework<V, C, A, N0> R0;
168   typedef cphstl::proxy_iterator<N0, R0> I0;
169   typedef cphstl::proxy_iterator<N0, R0, true> J0;
170   test_members<V, C, A, N0, R0, I0, J0>();
171   test_queue_sort<V, C, A, N0, R0, I0, J0>();
172
173   std::cout << "binary heaps passed\n";
174
175   typedef cphstl::element_encapsulator<V, std::ptrdiff_t, A> Nx;
176   typedef cphstl::weak_heap_heapifier Hx;
177   typedef cphstl::single_heap_framework<V, C, A, Nx, Hx> Rx;
178   typedef cphstl::proxy_iterator<Nx, Rx> Ix;
```

```
179   typedef cphstl::proxy_iterator<Nx, Rx, true> Jx;
180   test_members<V, C, A, Nx, Rx, Ix, Jx>();
181   test_queue_sort<V, C, A, Nx, Rx, Ix, Jx>();
182
183   std::cout << "weak heaps passed\n";
184
185   typedef cphstl::weak_heap_node<V, A> N1;
186   typedef cphstl::multiple_heap_framework<V, C, A, N1> R1;
187   typedef cphstl::priority_queue_iterator<N1, R1> I1;
188   typedef cphstl::priority_queue_iterator<N1, R1, true> J1;
189   test_members<V, C, A, N1, R1, I1, J1>();
190   test_queue_sort<V, C, A, N1, R1, I1, J1>();
191
192   std::cout << "weak queues passed\n";
193
194   typedef cphstl::pennant_node<V, A> N2;
195   typedef cphstl::multiple_heap_framework<V, C, A, N2> R2;
196   typedef cphstl::priority_queue_iterator<N2, R2> I2;
197   typedef cphstl::priority_queue_iterator<N2, R2, true> J2;
198   test_members<V, C, A, N2, R2, I2, J2>();
199   test_queue_sort<V, C, A, N2, R2, I2, J2>();
200
201   std::cout << "pennants queues passed\n";
202
203   typedef cphstl::fat_weak_heap_node<V, A> N3;
204   typedef cphstl::multiple_heap_framework<V, C, A, N3> R3;
205   typedef cphstl::priority_queue_iterator<N3, R3> I3;
206   typedef cphstl::priority_queue_iterator<N3, R3, true> J3;
207   test_members<V, C, A, N3, R3, I3, J3>();
208   test_queue_sort<V, C, A, N3, R3, I3, J3>();
209
210   std::cout << "weak queues with fat nodes passed\n";
211
212   typedef cphstl::fat_weak_heap_node<V, A> N4;
213   typedef cphstl::proxy_list_heap_store<C, A, N4> H4;
214   typedef cphstl::lazy_mark_store<C, A, N4> M4;
215   typedef cphstl::multiple_heap_framework<V, C, A, N4, H4, M4> R4;
216   typedef cphstl::priority_queue_iterator<N4, R4> I4;
217   typedef cphstl::priority_queue_iterator<N4, R4, true> J4;
218   test_members<V, C, A, N4, R4, I4, J4>();
219   test_queue_sort<V, C, A, N4, R4, I4, J4>();
220
221   std::cout << "run-relaxed weak queues passed\n";
222
223 #define RANK
224
225   typedef cphstl::fat_weak_heap_node<V, A> N5;
226   typedef cphstl::proxy_list_heap_store<C, A, N5> H5;
227   typedef cphstl::eager_mark_store<C, A, N5> M5;
228   typedef cphstl::multiple_heap_framework<V, C, A, N5, H5, M5> R5;
229   typedef cphstl::priority_queue_iterator<N5, R5> I5;
230   typedef cphstl::priority_queue_iterator<N5, R5, true> J5;
231   test_members<V, C, A, N5, R5, I5, J5>();
232   test_queue_sort<V, C, A, N5, R5, I5, J5>();
233
234   std::cout << "rank-relaxed weak queues passed\n";
235
236 #undef RANK
237
238   typedef cphstl::weak_heap_node<V, A> N6;
239   typedef cphstl::proxy_array_heap_store<C, A, N6> H6;
240   typedef cphstl::multiple_heap_framework<V, C, A, N6, H6> R6;
241   typedef cphstl::priority_queue_iterator<N6, R6> I6;
242   typedef cphstl::priority_queue_iterator<N6, R6, true> J6;
243   test_members<V, C, A, N6, R6, I6, J6>();
```

```
244   test_queue_sort<V, C, A, N6, R6, I6, J6>();
245
246   std::cout << "weak queues (proxy-array heap store) passed\n";
247
248   typedef cphstl::fat_weak_heap_node<V, A> N7;
249   typedef cphstl::proxy_list_heap_store<C, A, N7> H7;
250   typedef cphstl::simple_mark_store<C, A, N7> M7;
251   typedef cphstl::multiple_heap_framework<V, C, A, N7, H7, M7> R7;
252   typedef cphstl::priority_queue_iterator<N7, R7> I7;
253   typedef cphstl::priority_queue_iterator<N7, R7, true> J7;
254   test_members<V, C, A, N7, R7, I7, J7>();
255   test_queue_sort<V, C, A, N7, R7, I7, J7>();
256
257   std::cout << "simplified rank-relaxed weak queues passed\n";
258
259
260   return (0);
261 }
```

12.3 Meldable-priority-queue/Test/test-case.c++

```
 1 #include <algorithm>
 2 #include "assert.h++"
 3 #include <functional> // std::less
 4 #include <memory> // std::allocator
 5 #include <iostream>
 6 #include <vector>
 7
 8 template <typename V, typename Q>
 9 void test_case(int const numberofelements) {
10   V* a = new V[numberofelements];
11   for (int i = 0; i < numberofelements; i++) {
12     a[i] = i;
13   }
14   srand(1);
15   for (int i=0; i < numberofelements; i++) {
16     std::swap(a[i], a[rand() % (numberofelements)]);
17   }
18
19   Q q;
20   std::cout << "  insert ..." << "\n";
21   std::vector<typename Q::iterator> v;
22   for (int i = 0; i != numberofelements; ++i) {
23     typename Q::iterator p = q.push(a[i]);
24     v.push_back(p);
25   }
26
27   delete[] a;
28
29   std::cout << "  increase ..." << "\n";
30   for (int i = 0; i != numberofelements; ++i) {
31     typename Q::iterator p = v[i];
32     V value = *p + numberofelements;
33     //     std::cout << "increase: " << *p << "->" << value << "\n";
34     q.increase(p, value);
35     //     q.show();
36   }
37
38   std::cout << "  delete ..." << "\n";
39   for (int i = 0; i != numberofelements / 2; ++i) {
40     typename Q::iterator p = v[i];
41     //     std::cout << "delete: " << *p << "\n";
42     q.erase(p);
43     //     q.show();
44   }
```

```
45
46   std::cout << "  delete-min ..." << "\n";
47   while (! q.empty()) {
48     std::cout << "delete-min: " << *q.top() << "\n";
49     q.show();
50     q.pop();
51   }
52   assert(q.size() == 0);
53 }
54
55 #include <cstddef> // std::size_t
56 #include "element-encapsulator.h++"
57 #include "fat-weak-heap-node.h++"
58 #include "lazy-mark-store.h++"
59 #include <memory>
60 #include "multiple-heap-framework.h++"
61 #include "pennant-node.h++"
62 #include "proxy-iterator.h++"
63 #include "proxy-list-heap-store.h++"
64 #include "priority-queue-iterator.h++"
65 #include "simple-mark-store.h++"
66 #include "single-heap-framework.h++"
67 #include "stl-meldable-priority-queue.h++"
68 #include "weak-heap-heapifier.h++"
69 #include "weak-heap-node.h++"
70
71 int main() {
72
73   int const numberofelements = NUMBER;
74   std::cout << "\nn = " << numberofelements << "\n";
75
76   typedef long long V;
77   typedef std::less<V> C;
78   typedef std::allocator<V> A;
79
80   typedef cphstl::weak_heap_node<V, A> N1;
81   typedef cphstl::meldable_priority_queue<V, C, A, N1> Q1;
82   std::cout << "weak queue ...\n";
83   test_case<V, Q1>(numberofelements);
84
85   typedef cphstl::pennant_node<V, A> N2;
86   typedef cphstl::meldable_priority_queue<V, C, A, N2> Q2;
87   std::cout << "pennant queue ...\n";
88   test_case<V, Q2>(numberofelements);
89
90   typedef cphstl::fat_weak_heap_node<V, A> N3;
91   typedef cphstl::proxy_list_heap_store<C, A, N3> H3;
92   typedef cphstl::lazy_mark_store<C, A, N3> M3;
93   typedef cphstl::multiple_heap_framework<V, C, A, N3, H3, M3> R3;
94   typedef cphstl::priority_queue_iterator<N3, R3> I3;
95   typedef cphstl::priority_queue_iterator<N3, R3, true> J3;
96   typedef cphstl::meldable_priority_queue<V, C, A, N3, R3, I3, J3> Q3;
97   std::cout << "run-relaxed weak queue ...\n";
98   test_case<V, Q3>(numberofelements);
99
100  typedef cphstl::element_encapsulator<V, std::size_t, A> N4;
101  typedef cphstl::single_heap_framework<V, C, A, N4> R4;
102  typedef cphstl::proxy_iterator<N4, R4> I4;
103  typedef cphstl::proxy_iterator<N4, R4, true> J4;
104  typedef cphstl::meldable_priority_queue<V, C, A, N4, R4, I4, J4> Q4;
105  std::cout << "binary heap ...\n";
106  test_case<V, Q4>(numberofelements);
107
108  typedef cphstl::element_encapsulator<V, std::size_t, A> N5;
109  typedef cphstl::weak_heap_heapifier H5;
```

```
110   typedef cphstl::single_heap_framework<V, C, A, N5, H5> R5;
111   typedef cphstl::proxy_iterator<N5, R5> I5;
112   typedef cphstl::proxy_iterator<N5, R5, true> J5;
113   typedef cphstl::meldable_priority_queue<V, C, A, N5, R5, I5, J5> Q5;
114   std::cout << "weak heap ...\n";
115   test_case<V, Q5>(numberofelements);
116 
117   /*
118   typedef cphstl::fat_weak_heap_node<V, A> N6;
119   typedef cphstl::proxy_list_heap_store<C, A, N6> H6;
120   typedef cphstl::simple_mark_store<C, A, N6> M6;
121   typedef cphstl::multiple_heap_framework<V, C, A, N6, H6, M6> R6;
122   typedef cphstl::priority_queue_iterator<N6, R6> I6;
123   typedef cphstl::priority_queue_iterator<N6, R6, true> J6;
124   typedef cphstl::meldable_priority_queue<V, C, A, N6, R6, I6, J6> Q6;
125   std::cout << "simplified rank-relaxed weak queue ...\n";
126   test_case<V, Q6>(numberofelements);
127   */
128 
129   return 0;
130 }
```

12.4 Meldable-priority-queue/Test/unittest.mk

```
 1 CXXFLAGS = -Wall -std=c++0x -pedantic -x c++ -g -DDEBUG
 2 IFLAGS = -I $(HOME)/CPHSTL/Source/Assert/Code -I $(HOME)/CPHSTL/Source/Common/Code -
     I ../Code -I $(HOME)/CPHSTL/Source/Priority-queue-frameworks/Code -I $(HOME)/
     CPHSTL/Source/Iterator/Code -I $(HOME)/CPHSTL/Source/Proxy/Code -I $(HOME)/
     CPHSTL/Source/Type/Code -I .
 3 CXX = g++
 4 
 5 unit_tests: smoke-test larger-test comparative-test
 6 
 7 smoke-test:
 8         $(CXX) $(CXXFLAGS) $(IFLAGS) smoke-test.c++
 9         ./a.out
10         @rm -f a.out
11 
12 list = 23# 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
     29 30 31 32 33 34 35 36 37 38 39 40 41 99 100 101 #1000 10000 100000 1000000
13 larger-test:
14         for x in $(list) ; do \
15           $(CXX) $(CXXFLAGS) -DNUMBER=$$x $(IFLAGS) test-case.c++;\
16           ./a.out #> /dev/null; \
17 #         rm -f ./a.out ; \
18         done
19 
20 comparative-test:
21         for x in $(list) ; do \
22           $(CXX) $(CXXFLAGS) -DNUMBER=$$x $(IFLAGS) comparative-test.c++;\
23           ./a.out #> /dev/null; \
24 #         rm -f ./a.out ; \
25         done
26 
27 leak-check:
28         $(CXX) $(CXXFLAGS) $(IFLAGS) smoke-test.c++
29         valgrind --leak-check=full --show-reachable=yes ./a.out
30 
31 another-leak-check:
32         $(CXX) $(CXXFLAGS) -DNUMBER=10 $(IFLAGS) test-case.c++
33         valgrind --leak-check=full --show-reachable=yes ./a.out
34 
35 clean:
36         @rm -vf *~ a.out temp core
```

13. Benchmarks

13.1 Priority-queue-frameworks/Benchmark/push-comp.c++

```
1 #include <algorithm>
2 #include <ctime>
3 #include <iostream>
4
5 #ifndef NUMBER
6 #define NUMBER 1000000
7 #endif
8
9 #include "data-structure.i++" // Q and V come from here
10
11 int main() {
12   int const number_of_elements = NUMBER;
13
14   V* a = new V[number_of_elements];
15   for (int i = 0; i < number_of_elements; i++) {
16     a[i] = (V) i;
17   }
18
19   Q q;
20   for (int i = 0; i != number_of_elements; ++i) {
21     (void) q.push(a[i]);
22   }
23
24   double comp_count = double(comps);
25   double comp_per_operation = comp_count / double(number_of_elements);
26   std::cout << number_of_elements << " " << comp_per_operation << "\n";
27
28   q.clear();
29   delete[] a;
30   return 0;
31 }
```

13.2 Priority-queue-frameworks/Benchmark/increase-comp.c++

```
1 #include <algorithm>
2 #include <cmath>
3 #include <iostream>
4 #include <vector>
5
6 #ifndef NUMBER
7 #define NUMBER 1000000
8 #endif
9
10 #include "data-structure.i++" // Q and V come from here
11
12 int main() {
13   int const number_of_elements = NUMBER;
14
15   V* a = new V[number_of_elements];
16   for (int i = 0; i < number_of_elements; i++) {
17     a[i] = (V) i;
18   }
19   srand(1);
20   for (int i = 0; i < number_of_elements; i++) {
21     std::swap(a[i], a[rand() % (number_of_elements)]);
22   }
23
24   Q q;
25   std::vector<Q::iterator> v(number_of_elements);
26   for (int i = 0; i != number_of_elements; ++i) {
```

```
27      Q::iterator p = q.push(a[i]);
28      v[a[i]] = p;
29    }
30
31    comps = 0;
32    for (int i = 0; i != number_of_elements; ++i) {
33      q.increase(v[i], i + number_of_elements);
34    }
35
36    double comp_count = double(comps);
37    double comp_per_operation = comp_count / double(number_of_elements);
38    // double factor = comp_per_operation / std::log2(double(number_of_elements));
39    std::cout << number_of_elements << " " << comp_per_operation << "\n";
40    // std::cout << "(" << factor << " lg n)\n";
41
42    q.clear();
43    v.clear();
44    delete[] a;
45    return 0;
46  }
```

13.3 Priority-queue-frameworks/Benchmark/erase-comp.c++

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

```
  v.clear();
  delete[] a;
  return 0;
}
```

13.4 Priority-queue-frameworks/Benchmark/pop-comp.c++

```
#include <algorithm>
#include <cmath>
#include <iostream>

#ifndef NUMBER
#define NUMBER 1000000
#endif

#include "data-structure.i++" // Q and V come from here

int main() {
  int const number_of_elements = NUMBER;

  V* a = new V[number_of_elements];
  for (int i = 0; i < number_of_elements; i++) {
    a[i] = (V) i;
  }
  srand(1);
  for (int i = 0; i < number_of_elements; i++) {
    std::swap(a[i], a[rand() % (number_of_elements)]);
  }

  Q q;
  for (int i = 0; i != number_of_elements; ++i) {
    (void) q.push(a[i]);
  }

  comps = 0;
  for (int i = 0; i != number_of_elements; ++i) {
    q.pop();
  }

  double comp_count = double(comps);
  double comp_per_operation = comp_count / double(number_of_elements);
  // double factor = comp_per_operation / std::log2(double(number_of_elements));
  std::cout << number_of_elements << " " << comp_per_operation << "\n";
  // std::cout << "(" << factor << " lg n)\n";

  q.clear();
  delete[] a;
  return 0;
}
```

13.5 Priority-queue-frameworks/Benchmark/push-time.c++

```
#include <algorithm>
#include <ctime>
#include <iostream>

#ifndef NUMBER
#define NUMBER 1000000
#endif

#include "data-structure.i++" // Q and V come from here

int main() {
  int const number_of_elements = NUMBER;
  int repetitions = 1000000 / number_of_elements;
```

```
14   if (repetitions < 1) {
15     repetitions = 1;
16   }
17
18   V* a = new V[number_of_elements];
19
20   for (int i = 0; i < number_of_elements; i++) {
21     a[i] = (V) i;
22   }
23
24   Q* many = new Q[repetitions];
25   std::clock_t start = std::clock();
26   for (int k = 0; k != repetitions; ++k) {
27     for (int i = 0; i != number_of_elements; ++i) {
28       (void) many[k].push(a[i]);
29     }
30   }
31   std::clock_t stop = std::clock();
32   for (int k = 0; k != repetitions; ++k) {
33     many[k].clear();
34   }
35   delete[] many;
36   delete[] a;
37
38   double running_time = double(stop - start)/double(CLOCKS_PER_SEC);
39   double time_per_run = 1000000.0 * running_time / double(repetitions);
40   double time_per_operation = time_per_run / double(number_of_elements);
41   std::cout << number_of_elements << " " << time_per_operation << "\n";
42
43   return 0;
44 }
```

13.6 Priority-queue-frameworks/Benchmark/increase-time.c++

```
 1 #include <algorithm>
 2 #include <ctime>
 3 #include <iostream>
 4 #include <vector>
 5
 6 #ifndef NUMBER
 7 #define NUMBER 1000000
 8 #endif
 9
10 #include "data-structure.i++" // Q and V come from here
11
12 int main() {
13   int const number_of_elements = NUMBER;
14   int repetitions = 1000000 / number_of_elements;
15   if (repetitions < 1) {
16     repetitions = 1;
17   }
18
19   V* a = new V[number_of_elements];
20   for (int i = 0; i < number_of_elements; i++) {
21     a[i] = (V) i;
22   }
23
24   srand(1);
25   for (int i = 0; i < number_of_elements; i++) {
26     std::swap(a[i], a[rand() % (number_of_elements)]);
27   }
28
29   Q* many = new Q[repetitions];
30   std::clock_t start = std::clock();
31   for (int k = 0; k != repetitions; ++k) {
```

```
32      std::vector<Q::iterator> v(number_of_elements);
33      for (int i = 0; i != number_of_elements; ++i) {
34        Q::iterator p = many[k].push(a[i]);
35        v[a[i]] = p;
36      }
37      v.clear();
38    }
39    std::clock_t stop = std::clock();
40    double dual_time = double(stop - start)/double(CLOCKS_PER_SEC);
41
42    start = std::clock();
43    for (int k = 0; k != repetitions; ++k) {
44      std::vector<Q::iterator> v(number_of_elements);
45      for (int i = 0; i != number_of_elements; ++i) {
46        Q::iterator p = many[k].push(a[i]);
47        v[a[i]] = p;
48      }
49      for (int i = 0; i != number_of_elements; ++i) {
50        many[k].increase(v[i], i + number_of_elements);
51      }
52      v.clear();
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(number_of_elements);
62    std::cout << number_of_elements << " " << time_per_operation << "\n";
63
64    return 0;
65  }
```

13.7 Priority-queue-frameworks/Benchmark/erase-time.c++

```
1  #include <algorithm>
2  #include <ctime>
3  #include <iostream>
4  #include <vector>
5
6  #ifndef NUMBER
7  #define NUMBER 1000000
8  #endif
9
10 #include "data-structure.i++" // Q and V come from here
11
12 int main() {
13   int const number_of_elements = NUMBER;
14   int repetitions = 1000000 / number_of_elements;
15   if (repetitions < 1) {
16     repetitions = 1;
17   }
18
19   V* a = new V[number_of_elements];
20
21   for (int i = 0; i < number_of_elements; i++) {
22     a[i] = (V) i;
23   }
24
25   srand(1);
26   for (int i = 0; i < number_of_elements; i++) {
27     std::swap(a[i], a[rand() % (number_of_elements)]);
28   }
```

```
29
30   Q* many = new Q[repetitions];
31   std::clock_t start = std::clock();
32   for (int k = 0; k != repetitions; ++k) {
33     std::vector<Q::iterator> v;
34     for (int i = 0; i != number_of_elements; ++i) {
35       Q::iterator p = many[k].push(a[i]);
36       v.push_back(p);
37     }
38     v.clear();
39   }
40   std::clock_t stop = std::clock();
41   double dual_time = double(stop - start)/double(CLOCKS_PER_SEC);
42
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     std::vector<Q::iterator> v;
50     for (int i = 0; i != number_of_elements; ++i) {
51       Q::iterator p = many[k].push(a[i]);
52       v.push_back(p);
53     }
54     for (int i = 0; i != number_of_elements; ++i) {
55       many[k].erase(v[i]);
56     }
57     v.clear();
58   }
59   stop = std::clock();
60   delete[] many;
61   delete[] a;
62
63   double running_time = double(stop - start)/double(CLOCKS_PER_SEC);
64   running_time -= dual_time;
65   double time_per_run = 1000000.0 * running_time / double(repetitions);
66   double time_per_operation = time_per_run / double(number_of_elements);
67   std::cout << number_of_elements << " " << time_per_operation << "\n";
68
69   return 0;
70 }
```

13.8 Priority-queue-frameworks/Benchmark/pop-time.c++

```
 1 #include <algorithm>
 2 #include <ctime>
 3 #include <iostream>
 4
 5 #ifndef NUMBER
 6 #define NUMBER 1000000
 7 #endif
 8
 9 #include "data-structure.i++" // Q and V come from here
10
11 int main() {
12   int const number_of_elements = NUMBER;
13   int repetitions = 1000000 / number_of_elements;
14   if (repetitions < 1) {
15     repetitions = 1;
16   }
17
18   Q q;
19   V* a = new V[number_of_elements];
20
```

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

13.9 Priority-queue-frameworks/Benchmark/first-version.i++

```
1 #define FIRST_VERSION
2 #include "stl-meldable-priority-queue.h++"
3
4 long long comps = 0;
5
6 template <typename T>
7 class counting_comparator {
8 public:
9
10   bool operator()(T const& a, T const& b) const {
11     ++comps;
12     return a < b;
13   }
14 };
15
16 typedef long long V;
17 typedef counting_comparator<V> C;
18 typedef cphstl::meldable_priority_queue<V, C> Q;
```

13.10 Priority-queue-frameworks/Benchmark/binary-heap.i++

```
1 #include <cstddef> // std::size_t
2 #include "element-encapsulator.h++"
3 #include <memory> // std::allocator
4 #include "proxy-iterator.h++"
5 #include "single-heap-framework.h++"
6 #include "stl-meldable-priority-queue.h++"
7 #include <vector>
8
9 long long comps = 0;
10
11 template <typename T>
12 class counting_comparator {
13 public:
14
15   bool operator()(T const& a, T const& b) const {
16     ++comps;
17     return a < b;
18   }
19 };
20
21 typedef long long V;
22 typedef counting_comparator<V> C;
23 typedef std::allocator<V> A;
24 typedef cphstl::element_encapsulator<V, std::size_t, A> N;
25 typedef cphstl::single_heap_framework<V, C, A, N> R;
26 typedef cphstl::proxy_iterator<N, R> I;
27 typedef cphstl::proxy_iterator<N, R, true> J;
28 typedef cphstl::meldable_priority_queue<V, C, A, N, R, I, J> Q;
```

13.11 Priority-queue-frameworks/Benchmark/weak-heap.i++

```
1 #include <cstddef> // std::size_t
2 #include "element-encapsulator.h++"
3 #include <memory> // std::allocator
4 #include "proxy-iterator.h++"
5 #include "single-heap-framework.h++"
6 #include "stl-meldable-priority-queue.h++"
7 #include <vector>
8 #include "weak-heap-heapifier.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 V;
23 typedef counting_comparator<V> C;
24 typedef std::allocator<V> A;
25 typedef cphstl::element_encapsulator<V, std::ptrdiff_t, A> N;
26 typedef cphstl::weak_heap_heapifier H;
27 typedef cphstl::single_heap_framework<V, C, A, N, H> R;
28 typedef cphstl::proxy_iterator<N, R> I;
29 typedef cphstl::proxy_iterator<N, R, true> J;
30 typedef cphstl::meldable_priority_queue<V, C, A, N, R, I, J> Q;
```

13.12 Priority-queue-frameworks/Benchmark/weak-queue.i++

```
1 #include "stl-meldable-priority-queue.h++"
2
3 long long comps = 0;
4
5 template <typename T>
6 class counting_comparator {
7 public:
8
9   bool operator()(T const& a, T const& b) const {
10    ++comps;
11    return a < b;
12  }
13 };
14
15 typedef long long V;
16 typedef counting_comparator<V> C;
17 typedef cphstl::meldable_priority_queue<V, C> Q;
```

13.13 Priority-queue-frameworks/Benchmark/run-relaxed-weak-queue.i++

```
1 #include "proxy-list-heap-store.h++"
2 #include "fat-weak-heap-node.h++"
3 #include "lazy-mark-store.h++"
4 #include <memory>
5 #include "multiple-heap-framework.h++"
6 #include "priority-queue-iterator.h++"
7 #include "stl-meldable-priority-queue.h++"
8
9 long long comps = 0;
10
11 template <typename T>
12 class counting_comparator {
13 public:
14
15   bool operator()(T const& a, T const& b) const {
16     ++comps;
17     return a < b;
18   }
19 };
20
21 typedef long long V;
22 typedef counting_comparator<V> C;
23 typedef std::allocator<V> A;
24 typedef cphstl::fat_weak_heap_node<V, A> N;
25 typedef cphstl::proxy_list_heap_store<C, A, N> H;
26 typedef cphstl::lazy_mark_store<C, A, N> M;
27 typedef cphstl::multiple_heap_framework<V, C, A, N, H, M> R;
28 typedef cphstl::priority_queue_iterator<N, R> I;
29 typedef cphstl::priority_queue_iterator<N, R, true> J;
30 typedef cphstl::meldable_priority_queue<V, C, A, N, R, I, J> Q;
```

13.14 Priority-queue-frameworks/Benchmark/rank-relaxed-weak-queue.i++

```
1 #include "eager-mark-store.h++"
2 #include "fat-weak-heap-node.h++"
3 #include <memory>
4 #include "multiple-heap-framework.h++"
5 #include "priority-queue-iterator.h++"
6 #include "proxy-list-heap-store.h++"
7 #include "stl-meldable-priority-queue.h++"
8
```

```
9  long long comps = 0;
10
11 template <typename T>
12 class counting_comparator {
13 public:
14
15   bool operator()(T const& a, T const& b) const {
16     ++comps;
17     return a < b;
18   }
19 };
20
21 typedef long long V;
22 typedef counting_comparator<V> C;
23 typedef std::allocator<V> A;
24 typedef cphstl::fat_weak_heap_node<V, A> N;
25 typedef cphstl::proxy_list_heap_store<C, A, N> H;
26 typedef cphstl::eager_mark_store<C, A, N> M;
27 typedef cphstl::multiple_heap_framework<V, C, A, N, H, M> R;
28 typedef cphstl::priority_queue_iterator<N, R> I;
29 typedef cphstl::priority_queue_iterator<N, R, true> J;
30 typedef cphstl::meldable_priority_queue<V, C, A, N, R, I, J> Q;
```

13.15 Priority-queue-frameworks/Benchmark/fibonacci-heap.i++

```
1  #include <cstddef>
2  #include <LEDA/core/p_queue.h>
3  #include <LEDA/core/impl/f_heap.h>
4
5  long long comps = 0;
6
7  int compare(long long const& a, long long const& b) {
8   ++comps;
9    if (b < a) {
10     return -1;
11   }
12   if (a < b) {
13     return 1;
14   }
15   return 0;
16 } // max-heap order
17
18 class empty {
19 public:
20   empty() {
21   }
22 };
23
24 template <typename V, typename C, typename R>
25 class meldable_priority_queue_facade {
26 private:
27   R realizator;
28
29 public:
30
31   typedef V value_type;
32   typedef C comparator_type;
33   typedef typename R::item iterator;
34   typedef std::size_t size_type;
35
36   explicit meldable_priority_queue_facade(C cmp = compare)
37     : realizator(cmp) {
38   }
39
40   bool empty() const {
```

```
41      return realizator.empty();
42    }
43
44    size_type size() const {
45      return realizator.size();
46    }
47
48    iterator top() {
49      return realizator.find_min();
50    }
51
52    iterator push(V const& v) {
53      return realizator.insert(v, ::empty());
54    }
55
56    void pop() {
57      iterator p = realizator.find_min();
58      realizator.del_item(p);
59    }
60
61    void erase(iterator p) {
62      realizator.del_item(p);
63    }
64
65    void increase(iterator p, V const& v) {
66      realizator.decrease_p(p, v);
67    }
68
69    void clear () {
70      realizator.clear();
71    }
72  };
73
74  typedef long long V;
75  typedef int (*C) (V const&, V const&);
76  typedef leda::p_queue<V, empty, leda::f_heap> F;
77  typedef meldable_priority_queue_facade<V, C, F> Q;
```

13.16 Priority-queue-frameworks/Benchmark/pairing-heap.i++

```
1  #include <cstddef>
2  #include <LEDA/core/p_queue.h>
3  #include <LEDA/core/impl/p_heap.h>
4
5  long long comps = 0;
6
7  int compare(long long const& a, long long const& b) {
8    ++comps;
9    if (b < a) {
10     return -1;
11   }
12   if (a < b) {
13     return 1;
14   }
15   return 0;
16 } // max-heap order
17
18 class empty {
19 public:
20   empty() {
21   }
22 };
23
24 template <typename V, typename C, typename R>
25 class meldable_priority_queue_facade {
```

```
26 private:
27   R realizator;
28
29 public:
30
31   typedef V value_type;
32   typedef C comparator_type;
33   typedef typename R::item iterator;
34   typedef std::size_t size_type;
35
36   explicit meldable_priority_queue_facade(C cmp = compare)
37     : realizator(cmp) {
38   }
39
40   bool empty() const {
41     return realizator.empty();
42   }
43
44   size_type size() const {
45     return realizator.size();
46   }
47
48   iterator top() {
49     return realizator.find_min();
50   }
51
52   iterator push(V const& v) {
53     return realizator.insert(v, ::empty());
54   }
55
56   void pop() {
57     iterator p = realizator.find_min();
58     realizator.del_item(p);
59   }
60
61   void erase(iterator p) {
62     realizator.del_item(p);
63   }
64
65   void increase(iterator p, V const& v) {
66     realizator.decrease_p(p, v);
67   }
68
69   void clear () {
70     realizator.clear();
71   }
72 };
73
74 typedef long long V;
75 typedef int (*C) (V const&, V const&);
76 typedef leda::p_queue<V, empty, leda::p_heap> F;
77 typedef meldable_priority_queue_facade<V, C, F> Q;
```

13.17 Priority-queue-frameworks/Benchmark/benchmark.mk

```
1 CXXFLAGS = -DNDEBUG -Wall -std=c++0x -pedantic -x c++ -fno-strict-aliasing -O3
2 #CXXFLAGS = -DDEBUG -Wall -std=c++0x -pedantic -x c++ -g
3 IFLAGS = -I $(HOME)/CPHSTL/Source/Assert/Code -I $(HOME)/CPHSTL/Source/Meldable-
     priority-queue/Code -I ../Code -I $(HOME)/CPHSTL/Source/Priority-queue-
     frameworks/Code -I $(HOME)/CPHSTL/Source/Iterator/Code -I $(HOME)/CPHSTL/Source
     /Proxy/Code -I $(HOME)/CPHSTL/Source/Type/Code -I $(LEDAROOT)/incl -I . -L$(
     LEDAROOT)
4 LFLAGS = -lleda -lm
5 CXX = g++
6
```

```
 7 default:
 8         python oscilation.py >curves.log 2>&1
 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 push-time-tests:= $(addsuffix .push, $(time-tests))
15 push-comp-tests:= $(addsuffix .push, $(comp-tests))
16 increase-time-tests:= $(addsuffix .increase, $(time-tests))
17 increase-comp-tests:= $(addsuffix .increase, $(comp-tests))
18 erase-comp-tests:= $(addsuffix .erase, $(comp-tests))
19 erase-time-tests:= $(addsuffix .erase, $(time-tests))
20 pop-time-tests:= $(addsuffix .pop, $(time-tests))
21 pop-comp-tests:= $(addsuffix .pop, $(comp-tests))
22
23 $(time-tests): %.time: %.i++
24         @make -s $*.time.push
25         @make -s $*.time.increase
26         @make -s $*.time.erase
27         @make -s $*.time.pop
28
29 $(comp-tests): %.comp: %.i++
30         @make -s $*.comp.push
31         @make -s $*.comp.increase
32         @make -s $*.comp.erase
33         @make -s $*.comp.pop
34
35 list = 10000 100000 1000000
36
37 $(push-time-tests): %.time.push : %.i++
38         @echo $* "time per push"
39         @cp $*.i++ data-structure.i++
40         @for x in $(list) ; do \
41           $(CXX) $(CXXFLAGS) -DNUMBER=$$x $(IFLAGS) push-time.c++ $(LFLAGS);\
42           ./a.out; \
43           rm -f ./a.out ; \
44         done
45
46 $(increase-time-tests): %.time.increase : %.i++
47         @echo $* "time per increase"
48         @cp $*.i++ data-structure.i++
49         @for x in $(list) ; do \
50           $(CXX) $(CXXFLAGS) -DNUMBER=$$x $(IFLAGS) increase-time.c++ $(LFLAGS);\
51           ./a.out; \
52 #         rm -f ./a.out ; \
53         done
54         @rm data-structure.i++
55
56 $(erase-time-tests): %.time.erase : %.i++
57         @echo $* "time per erase"
58         @cp $*.i++ data-structure.i++
59         @for x in $(list) ; do \
60           $(CXX) $(CXXFLAGS) -DNUMBER=$$x $(IFLAGS) erase-time.c++ $(LFLAGS);\
61           ./a.out; \
62           rm -f ./a.out ; \
63         done
64         @rm data-structure.i++
65
66 $(pop-time-tests): %.time.pop : %.i++
67         @echo $* "time per pop"
68         @cp $*.i++ data-structure.i++
69         @for x in $(list) ; do \
70           $(CXX) $(CXXFLAGS) -DNUMBER=$$x $(IFLAGS) pop-time.c++ $(LFLAGS);\
71           ./a.out; \
```

```
72          rm -f ./a.out ; \
73        done
74        @rm data-structure.i++
75
76 $(push-comp-tests): %.comp.push : %.i++
77        @echo $* "#comp per push"
78        @cp $*.i++ data-structure.i++
79        @for x in $(list) ; do \
80          $(CXX) $(CXXFLAGS) -DNUMBER=$$x $(IFLAGS) push-comp.c++ $(LFLAGS);\
81          ./a.out; \
82          rm -f ./a.out ; \
83        done
84        @rm data-structure.i++
85
86 $(increase-comp-tests): %.comp.increase : %.i++
87        @echo $* "#comp per increase"
88        @cp $*.i++ data-structure.i++
89        @for x in $(list) ; do \
90          $(CXX) $(CXXFLAGS) -DNUMBER=$$x $(IFLAGS) increase-comp.c++ $(LFLAGS);\
91          ./a.out; \
92          rm -f ./a.out ; \
93        done
94        @rm data-structure.i++
95
96 $(erase-comp-tests): %.comp.erase : %.i++
97        @echo $* "#comp per erase"
98        @cp $*.i++ data-structure.i++
99        @for x in $(list) ; do \
100         $(CXX) $(CXXFLAGS) -DNUMBER=$$x $(IFLAGS) erase-comp.c++ $(LFLAGS);\
101         ./a.out; \
102         rm -f ./a.out ; \
103       done
104       @rm data-structure.i++
105
106 $(pop-comp-tests): %.comp.pop : %.i++
107       @echo $* "#comp per pop"
108       @cp $*.i++ data-structure.i++
109       @for x in $(list) ; do \
110         $(CXX) $(CXXFLAGS) -DNUMBER=$$x $(IFLAGS) pop-comp.c++ $(LFLAGS);\
111         ./a.out; \
112         rm -f ./a.out ; \
113       done
114       @rm data-structure.i++
115
116 clean:
117       @rm -vf *~ a.out temp plot.*
```

13.18 Priority-queue-frameworks/Benchmark/oscilation.py

```
1 """
2 Benchmarking meldable priority queues
3 """
4
5 __author__ = "Jyrki Katajainen"
6 __email__ = "jyrki@diku.dk"
7 __version__ = "$Revision: 1.9 $"[11:-2]
8 __date__ = "$Date: 2010-03-09 00:33:19 $"[7:-2]
9
10 import benz
11 import os
12
13 class case(benz.case):
14   def __init__(self, operation, measure, data_structure, n):
15     benz.case.__init__(self)
16     self.operation = operation
```

```
17     self.measure = measure
18     self.data_structure = data_structure
19     self.n = n
20     self.compiler = 'g++'
21     home = os.environ['HOME']
22     leda = os.environ['LEDAROOT']
23     self.compiler_options.extend(['-DNDEBUG', '-Wall', '-std=c++0x', '-pedantic', '-
         x␣c++', '-O3', '-DNUMBER=' + str(n)])
24     self.include_paths.extend([
25       home + '/CPHSTL/Source/Assert/Code/',
26       home + '/CPHSTL/Source/Meldable-priority-queue/Code/',
27       home + '/CPHSTL/Source/Priority-queue-frameworks/Code/',
28       home + '/CPHSTL/Source/Iterator/Code/',
29       home + '/CPHSTL/Source/Proxy/Code/',
30       home + '/CPHSTL/Source/Type/Code/',
31       leda + '/incl'])
32     compiler_options = '␣'.join(str(v) for v in self.compiler_options)
33     includes = '␣'.join('-I' + str(v) for v in self.include_paths)
34     libraries = '-L' + leda
35     file = operation + '-' + measure + '.c++'
36     copy = 'cp␣' + data_structure + '.i++␣data-structure.i++'
37     library_options = '-lleda␣-lm'
38     self.compile_command = copy + ';' + self.compiler + '␣' + compiler_options + '␣'
           + includes + '␣' + libraries + '␣' + file + '␣' + library_options
39     self.driver_file = './a.out'
40     clean = 'rm␣data-structure.i++'
41     self.execute_command = './a.out' + ';' + clean
42
43   def output(self):
44     if self.driver_output == '':
45       return ''
46     if self.driver_output == 'Segmentation␣fault':
47       return ''
48     (n, scalar) = self.driver_output.split('␣')
49     if float(scalar) < 0.0000001:
50       return ''
51     return (self.n, float(scalar))
52
53 class curve(benz.curve_suite):
54   def __init__(self, operation, measure, data_structure, title, linetype, pointtype,
         low, high):
55     benz.curve_suite.__init__(self)
56     self.title = title
57     self.linetype = linetype
58     self.pointtype = pointtype
59     for k in range(low, high + 1):
60       n = (1 << k)
61       delta = n / 8
62       for step in range(0, 8):
63         self.add(case(operation, measure, data_structure, n - 1 + step * delta))
64         self.add(case(operation, measure, data_structure, n + step * delta))
65
66 class plot(benz.plot_suite):
67   def __init__(self, operation, measure, low, high):
68     benz.plot_suite.__init__(self)
69     if operation == 'push':
70       op = '{/Times-Italic␣insert}'
71     elif operation == 'increase':
72       op = '{/Times-Italic␣decrease}'
73     elif operation == 'erase':
74       op = '{/Times-Italic␣delete}'
75     elif operation == 'pop':
76       op = '{/Times-Italic␣delete}-{/Times-Italic␣min}'
77     self.title = "Operation sequence: " + op + "^{/Times-Italic n}"
78     self.xlabel = '{/Times-Italic␣n}␣[logarithmic␣scale]'
```

```
79      if measure == 'comp':
80        self.ylabel = 'Number of element comparisons per {/Times-Italic n}'
81        self.gnuplot_commands += "set ytics 0,2\n"
82      if measure == 'time':
83        self.ylabel = 'Execution time per {/Times-Italic n} [{/Symbol m}s]'
84        self.gnuplot_commands += "set yrange [0.0:*]\n"
85      self.add(curve(operation, measure, 'binary-heap', 'binary heap', 'linetype 1', '
            pointtype 1', low, high))
86      self.add(curve(operation, measure, 'weak-heap', 'weak heap', 'linetype 2', '
            pointtype 2', low, high))
87      self.add(curve(operation, measure, 'weak-queue', 'weak queue', 'linetype 5', '
            pointtype 4', low, high))
88      self.add(curve(operation, measure, 'run-relaxed-weak-queue', 'run-relaxed weak 
            queue', 'linetype 9', 'pointtype 9', low, high))
89      self.add(curve(operation, measure, 'fibonacci-heap', 'Fibonacci heap', 'linetype
             3', 'pointtype 3', low, high))
90      self.add(curve(operation, measure, 'pairing-heap', 'pairing heap', 'linetype 7',
             'pointtype 6', low, high))
91      out_file = "set output '" + operation + '-' + measure + '.ps' + "'"
92      self.gnuplot_commands += out_file + """
93  set encoding iso_8859_1
94  set term postscript landscape enhanced color "Times" 14
95  set key left top Left reverse samplen 4 spacing 1.25 title ""
96  set data style linespoints
97  set title '%(title)s'
98  set xlabel '%(xlabel)s'
99  set ylabel '%(ylabel)s'
100 set logscale x
101 """ % self.__dict__
102     self.gnuplot_commands += "set format x '10^{%L}'\n"
103
104 class suite(benz.suite):
105   def __init__(self):
106     benz.suite.__init__(self)
107     self.add(plot('push', 'time', 10, 22))
108     self.add(plot('push', 'comp', 10, 22))
109     self.add(plot('increase', 'time',  10, 22))
110     self.add(plot('increase', 'comp',  10, 22))
111     self.add(plot('erase', 'time',  10, 22))
112     self.add(plot('erase', 'comp',  10, 22))
113     self.add(plot('pop', 'time',  10, 22))
114     self.add(plot('pop', 'comp',  10, 22))
115
116 if __name__ == '__main__':
117   benz.main(task = suite(), runner = benz.gnuplot_runner)
```