

Fat heaps: Source code

Amr Elmasry and Jyrki Katajainen

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

Abstract. This report is an electronic appendix to our paper “Fat heaps without regular counters”. In that paper we described a new variant of fat heaps that is conceptually simpler and easier to implement than the original version. We also compared the practical performance of this data structure to that of other related data structures (run-relaxed weak queues and Fibonacci heaps). This report together with an accompanying `tar` file gives the source code used in the experiments reported in the paper. By making the programs publicly available, we provide other researchers the opportunity to scrutinize the code and compare their own implementations against ours.

Keywords. Priority queues, fat heaps, worst-case efficiency

Copyright notice

Copyright © 2000–2011 by The authors and Performance Engineering Laboratory (University of Copenhagen)

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

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

Release date

2011-12-15

Included files

File	Page
§ 1 relaxed-heap.h++	4
§ 2 fat-heap-node.h++	7
§ 3 semi-sorted-tree-inventory.h++	12
§ 4 blank-violation-inventory.h++	15
§ 5 arena-based-violation-inventory.h++	16
§ 6 fat-heap-transformer.h++	20
§ 7 bit-store.h++	23
§ 8 bit-manipulation.h++	25
§ 9 meldable-priority-queue.h++	27
§ 10 meldable-priority-queue.i++	29
§ 11 node-iterator.h++	33
§ 12 node-iterator.i++	36
§ 13 comparator-proxy.h++	39
§ 14 top-decorator.h++	40
§ 15 fat-heap.i++	42
§ 16 slow-increase-fat-heap.i++	42
§ 17 fast-top-fat-heap.i++	43
§ 18 push-time.c++	43
§ 19 increase-time.c++	45
§ 20 erase-time.c++	46
§ 21 pop-time.c++	47
§ 22 push-comp.c++	48
§ 23 increase-comp.c++	50
§ 24 erase-comp.c++	51
§ 25 pop-comp.c++	51
§ 26 benchmark.mk	52

Kernel

§ 1 *relaxed-heap.h++*

```

1  /*
2  A relaxed-heap framework which can be used to implement, for
3  example, run-relaxed and fat heaps.
4
5  Operations to be supported by the tree inventory: default
6  constructor, empty, --first--, --next--, top, insert, extract,
7  reduce, swap, show, is_valid.
8
9  Operations to be supported by the violation inventory: default
10 constructor, empty, --first--, --next--, top, insert, extract,
11 reduce, swap, show, is_valid.
12
13 Author: Jyrki Katajainen © 2010
14 */
15
16 #ifndef __CPHSTL_RELAXED_HEAP__
17 #define __CPHSTL_RELAXED_HEAP__
18
19 #include <algorithm>
20 #include "comparator-proxy.h++"
21 #include <cstdlib> // std::size_t
22 #include <iostream>
23 #include <utility> // std::pair
24 #include <vector>
25
26 namespace cphstl {
27 template <typename E, typename C, typename N, typename T, typename V>
28 class relaxed_heap {
29 public:
30
31     // types
32
33     typedef E element_type;
34     typedef C comparator_type;
35     typedef N node_type;
36     typedef T tree_inventory_type;
37     typedef V violation_inventory_type;
38     typedef std::size_t size_type;
39     typedef E& reference;
40     typedef E const& const_reference;
41     typedef N* locator_type;
42     typedef N const* const_locator_type;
43
44 protected:
45
46     // variables
47
48     comparator_proxy<C> comparator;
49     T tree_inventory;
50     V violation_inventory;
51     size_type n;
52
53 public:
54
55     // structors
56
57     explicit relaxed_heap(C const& c = C())
58         : comparator(c), tree_inventory(c), violation_inventory(c), n(0) {
59     }
60

```

```

61     ~relaxed_heap() {
62     }
63
64     // iterators
65
66     N* begin() {
67         if (tree_inventory.empty()) {
68             return (N*) 0;
69         }
70         return tree_inventory.__first__();
71     }
72
73     N const* begin() const {
74         if (tree_inventory.empty()) {
75             return (N*) 0;
76         }
77         return tree_inventory.__first__();
78     }
79
80     N* end() {
81         return (N*) 0;
82     }
83
84     N const* end() const {
85         return (N*) 0;
86     }
87
88     // accessors
89
90     C get_comparator() const {
91         return C(comparator.subject());
92     }
93
94     size_type size() const {
95         return n;
96     }
97
98     size_type max_size() const {
99         typename std::vector<int>::allocator_type a;
100         size_type available_memory = a.max_size() * sizeof(int); // in bytes
101         return n + available_memory / sizeof(N);
102     }
103
104     N* top() const {
105         N* p = tree_inventory.top();
106         N* q = violation_inventory.top();
107         if (p == 0) {
108             return q;
109         }
110         if (q == 0 || comparator((*q).element(), (*p).element())) {
111             return p;
112         }
113         return q;
114     }
115
116     // modifiers
117
118     N* insert(N* p) {
119         tree_inventory.insert(p);
120         tree_inventory.reduce(violation_inventory);
121         ++n;
122         return p;
123     }
124
125     N* extract() {

```

```

126     N* root = tree_inventory.__first__();
127     tree_inventory.extract(root);
128     N* q = (*root).right();
129     while (q != 0) {
130         violation_inventory.extract(q);
131         N* r = (*q).left();
132         (*q).cut();
133         tree_inventory.insert(q);
134         q = r;
135     }
136     (*root).rank(0);
137     --n;
138     tree_inventory.reduce(violation_inventory);
139     violation_inventory.reduce(tree_inventory);
140     return root;
141 }
142
143 N* extract(N* p) {
144     N* replacement = extract();
145     if (p == replacement) {
146         return p;
147     }
148     violation_inventory.extract(p);
149     N* r = p;
150     N* s = (*p).right();
151     while (s != 0) {
152         violation_inventory.extract(s);
153         r = s;
154         s = (*r).left();
155     }
156     N* q = replacement;
157     if (r != p) {
158         (*r).left(replacement);
159         (*replacement).left(0);
160         (*replacement).parent(r);
161         while ((*q).parent() != p) {
162             q = (*q).join(comparator, tree_inventory, violation_inventory);
163         }
164         (*p).right(0);
165         (*q).parent(0);
166     }
167     (*p).replace(q, tree_inventory);
168     violation_inventory.insert(q);
169     violation_inventory.reduce(tree_inventory);
170     return p;
171 }
172
173 void increase(N* p, E const& v) {
174     (*p).element() = v;
175     violation_inventory.insert(p);
176     violation_inventory.reduce(tree_inventory);
177 }
178
179 void meld(relaxed_heap& other) {
180     if (this == &other) {
181         return;
182     }
183     if (size() < other.size()) {
184         swap(other);
185     }
186     while (! other.tree_inventory.empty()) {
187         N* t = other.tree_inventory.__first__();
188         other.tree_inventory.extract(t);
189         tree_inventory.insert(t);
190         tree_inventory.reduce(violation_inventory);

```

```

191     }
192     while (! other.violation_inventory.empty()) {
193         N* v = other.violation_inventory.__first__();
194         other.violation_inventory.extract(v);
195         violation_inventory.insert(v);
196         violation_inventory.reduce(tree_inventory);
197     }
198     n += other.n;
199     other.n = 0;
200 }
201
202 void swap(relaxed_heap & other) {
203     std::swap(comparator, other.comparator);
204     tree_inventory.swap(other.tree_inventory);
205     violation_inventory.swap(other.violation_inventory);
206     std::swap(n, other.n);
207 }
208
209 };
210 }
211
212 #endif

```

Node

§ 2 *fat-heap-node.h++*

```

1  /*
2  A node used in a fat heap; the binary-tree terminology is used all
3  over.
4
5  Author: Jyrki Katajainen © 2010
6  */
7
8  #ifndef __CPHSTL_FAT_HEAP_NODE__
9  #define __CPHSTL_FAT_HEAP_NODE__
10
11 #include <algorithm> // std::swap
12 #include <cstddef> // std::size_t
13 #include <list>
14 #include <memory> // std::allocator
15
16 namespace cphstl {
17     template <typename E, typename A = std::allocator<E> >
18     class fat_heap_node {
19     public:
20
21         typedef E element_type;
22         typedef A allocator_type;
23         typedef std::size_t size_type;
24         typedef unsigned char rank_type;
25         typedef unsigned char index_type;
26
27         rank_type rank_;
28         bool is_root_;
29         index_type violation_index_;
30         fat_heap_node* parent_;
31         fat_heap_node* left_child_;
32         fat_heap_node* right_child_;
33         E element_;
34
35     private:
36
37         fat_heap_node();

```

```

38     fat_heap_node(fat_heap_node const &);
39     fat_heap_node & operator=(fat_heap_node const &);
40
41     public:
42
43     fat_heap_node(E const & v, A const & = A())
44         : rank_(0), is_root_(true), violation_index_(0xff),
45           parent_(0), left_child_(0), right_child_(0), element_(v) {
46     }
47
48     rank_type rank() const {
49         return rank_;
50     }
51
52     void rank(rank_type r) {
53         rank_ = r;
54     }
55
56     bool is_root() const {
57         return is_root_;
58     }
59
60     void is_root(bool b) {
61         is_root_ = b;
62     }
63
64     bool is_marked() const {
65         return violation_index_ != 0xff;
66     }
67
68     index_type violation_index() const {
69         return violation_index_;
70     }
71
72     void violation_index(index_type i) {
73         violation_index_ = i;
74     }
75
76     fat_heap_node* parent() const {
77         return parent_;
78     }
79
80     void parent(fat_heap_node* p) {
81         parent_ = p;
82     }
83
84
85     fat_heap_node* left() const {
86         return left_child_;
87     }
88
89     void left(fat_heap_node* p) {
90         left_child_ = p;
91     }
92
93
94     fat_heap_node* right() const {
95         return right_child_;
96     }
97
98     void right(fat_heap_node* p) {
99         right_child_ = p;
100
101     }
102

```

```

103 E const& element() const {
104     return element_;
105 }
106
107 E& element() {
108     return element_;
109 }
110
111 void swap_attributes(fat_heap_node* q) {
112     fat_heap_node* p = this;
113     std::swap((*p).rank_, (*q).rank_);
114     std::swap((*p).is_root_, (*q).is_root_);
115 }
116
117 template <typename C, typename T, typename V>
118 fat_heap_node* join(C const& comparator, T&, V&) {
119     fat_heap_node* p = this;
120     fat_heap_node* q = (*p).parent();
121     fat_heap_node* r = (*q).parent();
122     fat_heap_node* s = (*r).parent();
123     if (comparator((*q).element(), (*p).element())) {
124         std::swap(q, p);
125     }
126     if (comparator((*r).element(), (*q).element())) {
127         std::swap(r, q);
128     }
129     fat_heap_node* c = (*r).right();
130     if (c != 0) {
131         (*c).parent(p);
132     }
133     (*p).left(c);
134     (*p).parent(q);
135     (*p).is_root(false);
136     (*q).left(p);
137     (*q).parent(r);
138     (*q).is_root(false);
139     (*r).left(0);
140     (*r).right(q);
141     (*r).rank((*r).rank() + 1);
142     (*r).parent(s);
143     return r;
144 }
145
146 void cut() { // parent's rank not updated
147     fat_heap_node* q = this;
148     fat_heap_node* p = (*q).parent();
149     fat_heap_node* r = (*q).left();
150     if ((*p).right() == q) {
151         (*p).right(r);
152     }
153     else {
154         (*p).left(r);
155     }
156     if (r != 0) {
157         (*r).parent(p);
158     }
159     (*q).parent(0);
160     (*q).left(0);
161     (*q).is_root(true);
162 }
163
164 template <typename T>
165 void replace(fat_heap_node* v, T& tree_inventory) {
166     fat_heap_node* u = this;
167     if ((*u).is_root()) {

```

```

168     tree_inventory.replace(u, v);
169 }
170 else {
171     (*u).swap_attributes(v);
172     fat_heap_node* t = (*u).left();
173     if (t != 0) {
174         (*t).parent(v);
175     }
176     (*v).left(t);
177     fat_heap_node* p = (*u).parent();
178     (*v).parent(p);
179     if ((*p).right() == u) {
180         (*p).right(v);
181     }
182     else {
183         (*p).left(v);
184     }
185 }
186 }
187
188 fat_heap_node* distinguished_ancestor() const {
189     fat_heap_node const* q = this;
190     if ((*q).is_root()) {
191         return (fat_heap_node*) 0;
192     }
193     fat_heap_node* p = (*q).parent();
194     while (! (*p).is_root() && (*p).left() == q) {
195         q = p;
196         p = (*p).parent();
197     }
198     return p;
199 }
200
201 template <typename T>
202 fat_heap_node* promote(fat_heap_node* p, T& tree_inventory) {
203     fat_heap_node* q = this;
204     fat_heap_node* a = (*p).parent();
205     bool at_root = (*p).is_root();
206     fat_heap_node* c = (*p).right();
207     fat_heap_node* e = (*q).parent();
208     fat_heap_node* f = (*q).left();
209     fat_heap_node* g = (*q).right();
210     if (p == (*q).parent()) {
211         if (at_root) {
212             tree_inventory.replace(p, q);
213         }
214         else {
215             (*p).swap_attributes(q);
216             fat_heap_node* b = (*p).left();
217             if ((*a).left() == p) {
218                 (*a).left(q);
219             }
220             else {
221                 (*a).right(q);
222             }
223             (*q).parent(a);
224             (*q).left(b);
225             if (b != 0) {
226                 (*b).parent(q);
227             }
228         }
229         (*p).parent(q);
230         (*p).left(f);
231         (*p).right(g);
232         (*q).right(p);

```

```

233     if (f != 0) {
234         (*f).parent(p);
235     }
236     if (g != 0) {
237         (*g).parent(p);
238     }
239 }
240 else {
241     if (at_root) {
242         tree_inventory.replace(p, q);
243     }
244     else {
245         (*p).swap_attributes(q);
246         fat_heap_node* b = (*p).left();
247         if ((*a).left() == p) {
248             (*a).left(q);
249         }
250         else {
251             (*a).right(q);
252         }
253         (*q).parent(a);
254         (*q).left(b);
255         if (b != 0) {
256             (*b).parent(q);
257         }
258     }
259     (*q).right(c);
260     (*c).parent(q);
261     (*p).parent(e);
262     (*e).left(p);
263     (*p).left(f);
264     (*p).right(g);
265     if (f != 0) {
266         (*f).parent(p);
267     }
268     if (g != 0) {
269         (*g).parent(p);
270     }
271 }
272 return q;
273 }
274
275 fat_heap_node const* successor() const {
276     fat_heap_node const* x = this;
277     if ((*x).right() != 0) {
278         x = (*x).right();
279         while ((*x).left() != 0) {
280             x = (*x).left();
281         }
282     }
283     return x;
284 }
285 fat_heap_node const* y = (*x).parent();
286 while (y != 0 && x == (*y).right()) {
287     x = y;
288     y = (*y).parent();
289 }
290 return y;
291 }
292 fat_heap_node const* root() const {
293     fat_heap_node const* p = this;
294     while (! (*p).is_root()) {
295         p = (*p).parent();
296     }
297     return p;

```

```

298     }
299
300   };
301 }
302
303 #endif

```

Tree inventory

§ 3 *semi-sorted-tree-inventory.h++*

```

1  /*
2  This inventory keeps trees of the same rank in doubly-linked lists
3  as described in [J.R. Driscoll et al., Relaxed heaps: An alternative
4  to Fibonacci heaps with applications to parallel computation,
5  Communications of the ACM 31,11 (1988), 1343-1354]
6
7  Author: Jyrki Katajainen © 2010
8  */
9
10 #ifndef __CPHSTL_SEMI_SORTED_TREE_INVENTORY__
11 #define __CPHSTL_SEMI_SORTED_TREE_INVENTORY__
12
13 #include <algorithm> // std::swap
14 #include "bit-store.h++"
15 #include <climits> // CHAR_BIT
16 #include "comparator-proxy.h++"
17 #include <cstdlib> // std::size_t
18 #include <iostream>
19 #include <vector>
20
21 namespace cphstl {
22   template<int threshold, typename C, typename N, typename W = unsigned long>
23   class semi_sorted_tree_inventory {
24   public:
25
26     typedef C comparator_type;
27     typedef N node_type;
28     typedef W word_type;
29     typedef std::size_t size_type;
30
31     enum {word_size = CHAR_BIT * sizeof(W)};
32
33   protected:
34
35     struct pair_type {
36       size_type count;
37       N* root;
38
39       pair_type(size_type c = 0, N* p = 0)
40         : count(c), root(p) {
41       }
42     };
43
44     comparator_proxy<C> comparator;
45     cphstl::bit_store<W> occupied;
46     cphstl::bit_store<W> saturated;
47     std::vector<pair_type> header;
48
49   public:
50
51     semi_sorted_tree_inventory(C const & c = C())
52       : comparator(c), occupied(0), saturated(0), header() {
53       header.resize(word_size);

```

```

54     }
55
56     ~semi_sorted_tree_inventory() {
57     }
58
59     bool empty() const {
60         return occupied.empty();
61     }
62
63     N* __first__() const {
64         if (occupied.empty()) {
65             return 0;
66         }
67         return header[occupied.least_significant_one()].root;
68     }
69
70     N* __next__(N* current) const {
71         if ((*current).parent() != 0) {
72             return (*current).parent();
73         }
74         size_type rank = (*current).rank();
75         cphstl::bit_store<W> copy = occupied;
76         copy.unset(size_type(0), rank + 1);
77         if (copy.empty()) {
78             return 0;
79         }
80         return header[copy.least_significant_one()].root;
81     }
82
83     N* top() const {
84         N* maximum = __first__();
85         if (maximum == 0) {
86             return maximum;
87         }
88         N* node = __next__(maximum);
89         while (node != 0) {
90             if (comparator((*maximum).element(), (*node).element())) {
91                 maximum = node;
92             }
93             node = __next__(node);
94         }
95         return maximum;
96     }
97
98     void insert(N* p) {
99         size_type rank = (*p).rank();
100        N* q = header[rank].root;
101        header[rank].count += 1;
102        header[rank].root = p;
103        (*p).left(0);
104        (*p).parent(q);
105        if (q != 0) {
106            (*q).left(p);
107        }
108        occupied.set(rank);
109        if (header[rank].count >= threshold) {
110            saturated.set(rank);
111        }
112    }
113
114    void extract(N* q) {
115        size_type rank = (*q).rank();
116        N* p = (*q).left();
117        N* r = (*q).parent();
118        header[rank].count -= 1;

```

```

119     if (q == header[rank].root) {
120         header[rank].root = r;
121     }
122     if (header[rank].count == 0) {
123         occupied.unset(rank);
124     }
125     if (header[rank].count < threshold) {
126         saturated.unset(rank);
127     }
128     if (p != 0) {
129         (*p).parent(r);
130     }
131     if (r != 0) {
132         (*r).left(p);
133     }
134     (*q).parent(0);
135     (*q).left(0);
136 }
137
138 void replace(N* p, N* q) {
139     extract(p);
140     (*p).swap_attributes(q);
141     insert(q);
142 }
143
144 template <typename V>
145 void reduce(V& violation_inventory) {
146     if (saturated.empty()) {
147         return;
148     }
149     size_type rank = saturated.least_significant_one();
150     N* p = header[rank].root;
151     N* q = (*p).join(comparator, *this, violation_inventory);
152     N* r = (*q).parent();
153     header[rank].count -= threshold;
154     header[rank].root = r;
155     (*q).parent(0);
156     if (r != 0) {
157         (*r).left(0);
158     }
159     if (header[rank].count == 0) {
160         occupied.unset(rank);
161     }
162     if (header[rank].count < threshold) {
163         saturated.unset(rank);
164     }
165     insert(q);
166 }
167
168 void swap(semi_sorted_tree_inventory& other) {
169     std::swap(comparator, other.comparator);
170     std::swap(occupied, other.occupied);
171     std::swap(saturated, other.saturated);
172     header.swap(other.header);
173 }
174
175 };
176 }
177
178 #endif

```

Violation inventories

§ 4 *blank-violation-inventory.h++*

```

1  /*
2  A blank violation inventory; a violation node must be removed before
3  any new violation nodes are inserted.
4
5  Author: Jyrki Katajainen © 2009, 2010
6  */
7
8  #ifndef __CPHSTL_BLANK_VIOLATION_INVENTORY__
9  #define __CPHSTL_BLANK_VIOLATION_INVENTORY__
10
11 #include <algorithm> // std::swap
12 #include "comparator-proxy.h++"
13 #include <cstddef> // std::size_t
14 #include <iostream>
15
16 namespace cphstl {
17     template <typename C, typename N>
18     class blank_violation_inventory {
19     public:
20
21         typedef C comparator_type;
22         typedef N node_type;
23         typedef std::size_t size_type;
24
25     protected:
26
27         comparator_proxy<C> comparator;
28         N* single_mark;
29
30     private:
31
32         blank_violation_inventory(blank_violation_inventory const &);
33         blank_violation_inventory & operator=(blank_violation_inventory const &);
34
35     public:
36
37         explicit blank_violation_inventory(C const & c = C())
38             : comparator(c), single_mark(0) {
39         }
40
41         ~blank_violation_inventory() {
42         }
43
44         bool empty() const {
45             return single_mark == 0;
46         }
47
48         N* __first__() const {
49             return single_mark;
50         }
51
52         N* top() const {
53             return single_mark;
54         }
55
56         void insert(N* p) {
57             if (! (*p).is_root()) {
58                 single_mark = p;
59             }
60         }

```

```

61
62 void extract(N* p) {
63     single_mark = (single_mark == p) ? 0 : single_mark;
64 }
65
66 template <typename T>
67 void reduce(T& tree_inventory) {
68     if (single_mark == 0 || (*single_mark).is_root()) {
69         single_mark = 0;
70         return;
71     }
72     N* q = single_mark;
73     N* p = (*q).distinguished_ancestor();
74     while (p != 0) {
75         if (comparator((*p).element(), (*q).element())) {
76             (*q).promote(p, tree_inventory);
77             p = (*q).distinguished_ancestor();
78         }
79         else {
80             break;
81         }
82     }
83     single_mark = 0;
84 }
85
86 template <typename T>
87 void meld(blank_violation_inventory& other, T& tree_inventory) {
88     if (other.single_mark != 0) {
89         other.reduce(tree_inventory);
90     }
91 }
92
93 void swap(blank_violation_inventory& other) {
94     std::swap(comparator, other.comparator);
95     std::swap(single_mark, other.single_mark);
96 }
97
98 };
99 }
100
101 #endif

```

§ 5 arena-based-violation-inventory.h++

```

1 /*
2  This inventory maintains references to the violation nodes in a
3  small contiguous segment of memory. By using our own memory manager,
4  cursors to these references only occupy  $O(\lg \lg n)$  bits (which can be
5  stored in a byte). This way the nodes and the overall memory
6  footprint of the underlying priority queue become smaller.
7
8  Author: Jyrki Katajainen © 2010
9  */
10
11 #ifndef __CPHSTL_ARENA_BASED_VIOLATION_INVENTORY__
12 #define __CPHSTL_ARENA_BASED_VIOLATION_INVENTORY__
13
14 #include <algorithm> // std::swap
15 #include "bit-store.h++"
16 #include <climits> // CHAR_BIT
17 #include "comparator-proxy.h++"
18 #include <cstddef> // std::size_t
19 #include <iostream>
20 #include <vector>

```

```

21
22 namespace cphstl {
23     template<typename C, typename N, typename S, typename W = unsigned long long>
24     class arena_based_violation_inventory {
25     public:
26
27         typedef C comparator_type;
28         typedef N node_type;
29         typedef S transformer_type;
30         typedef W word_type;
31         typedef unsigned char index_type;
32         typedef std::size_t size_type;
33
34         enum {word_size = CHAR_BIT * sizeof(W)};
35
36     protected:
37
38         struct list_node {
39             N* violation_node;
40             index_type next;
41             index_type previous;
42         };
43
44         comparator_proxy<C> comparator;
45         S transformer;
46         cphstl::bit_store<W> occupied;
47         cphstl::bit_store<W> saturated;
48         std::vector<N*> header;
49         cphstl::bit_store<W> free;
50         std::vector<list_node> segment;
51
52     public:
53
54         explicit arena_based_violation_inventory(C const& c = C())
55             : comparator(c), transformer(c), occupied(), saturated(), header(),
56             free(~W(0)), segment() {
57             header.resize(word_size, (N*) 0);
58             segment.resize(word_size);
59         }
60
61         ~arena_based_violation_inventory() {
62         }
63
64         bool empty() const {
65             return occupied.empty();
66         }
67
68         N* __first__() const {
69             if (occupied.empty()) {
70                 return (N*) 0;
71             }
72             size_type rank = occupied.least_significant_one();
73             return header[rank];
74         }
75
76         N* __next__(N* current) const {
77             index_type i = (*current).violation_index();
78             index_type j = segment[i].next;
79             return segment[j].violation_node; // Warning: violation list circular
80         }
81
82         N* top() const {
83             N* maximum = __first__();
84             if (maximum == 0) {
85                 return maximum;

```

```

86     }
87     N* node = __next__(maximum);
88     while (node != __first__()) {
89         if (comparator((*maximum).element(), (*node).element())) {
90             maximum = node;
91         }
92         node = __next__(node);
93     }
94     return maximum;
95 }
96
97 void insert(N* p) {
98     if ((*p).is_marked() || (*p).is_root()) {
99         return;
100     }
101     size_type rank = (*p).rank();
102     cphstl::bit_store<W> copy;
103     if (occupied.empty()) {
104         (*p).violation_index(add_first(p));
105         occupied.set(rank);
106     }
107     else if (! occupied.get(rank)) {
108         copy = occupied;
109         copy.unset(size_type(0), rank + 1);
110         if (copy.empty()) {
111             size_type smallest = occupied.least_significant_one();
112             (*p).violation_index(add_before(header[smallest], p));
113         }
114         else {
115             size_type nearest_larger = copy.least_significant_one();
116             (*p).violation_index(add_before(header[nearest_larger], p));
117         }
118         occupied.set(rank);
119     }
120     else if (! saturated.get(rank)) {
121         (*p).violation_index(add_before(header[rank], p));
122         saturated.set(rank);
123     }
124     else {
125         (*p).violation_index(add_before(header[rank], p));
126     }
127     header[rank] = p;
128 }
129
130 void extract(N* p) {
131     if (! (*p).is_marked()) {
132         return;
133     }
134     size_type rank = (*p).rank();
135     N* q;
136     N* r;
137     if (! occupied.get(rank)) {
138     }
139     else if (! saturated.get(rank)) {
140         cut(p);
141         header[rank] = (N*) 0;
142         occupied.unset(rank);
143     }
144     else {
145         if (header[rank] == p) {
146             q = __next__(p);
147             cut(p);
148             header[rank] = q;
149         }
150         else {

```

```

151         cut(p);
152         q = header[rank];
153     }
154     r = __next__(q);
155     if (r == header[rank] || (*r).rank() != rank) {
156         saturated.unset(rank);
157     }
158 }
159 }
160
161 template <typename T>
162 void reduce(T& tree_inventory) {
163     if (saturated.empty()) {
164         return;
165     }
166     size_type rank = saturated.most_significant_one();
167     N* p = header[rank];
168     N* q = __next__(p);
169     N* s = __next__(q);
170     cut(p);
171     cut(q);
172     if (s == p || (*s).rank() != rank) { // 0
173         header[rank] = (N*) 0;
174         occupied.unset(rank);
175         saturated.unset(rank);
176     }
177     else {
178         header[rank] = s;
179         N* t = __next__(s);
180         if (t == s || (*t).rank() != rank) { // 1
181             occupied.set(rank);
182             saturated.unset(rank);
183         }
184         else { // 2+
185             }
186     }
187     N* r = transformer.reduce(p, q, tree_inventory, *this);
188     insert(r);
189 }
190
191 void swap(arena_based_violation_inventory& other) {
192     std::swap(comparator, other.comparator);
193     std::swap(transformer, other.transformer);
194     std::swap(occupied, other.occupied);
195     std::swap(saturated, other.saturated);
196     header.swap(other.header);
197     std::swap(free, other.free);
198     segment.swap(other.segment);
199 }
200
201 protected:
202
203     index_type add_first(N* q) {
204         index_type f = free.least_significant_one();
205         free.unset(f);
206         segment[f].previous = f;
207         segment[f].violation_node = q;
208         segment[f].next = f;
209         return f;
210     }
211
212     index_type add_before(N* r, N* q) {
213         index_type j = free.least_significant_one();
214         free.unset(j);
215         index_type k = (*r).violation_index();

```

```

216     index_type i = segment[k].previous;
217     segment[i].next = j;
218     segment[j].previous = i;
219     segment[j].violation_node = q;
220     segment[j].next = k;
221     segment[k].previous = j;
222     return j;
223 }
224
225 void cut(N* q) {
226     index_type j = (*q).violation_index();
227     (*q).violation_index(0xff);
228     index_type i = segment[j].previous;
229     index_type k = segment[j].next;
230     segment[i].next = k;
231     segment[k].previous = i;
232     segment[j].previous = 0xff;
233     segment[j].violation_node = (N*) 0;
234     segment[j].next = 0xff;
235     free.set(j);
236 }
237 };
238 }
239
240 #endif

```

Transformations

§ 6 *fat-heap-transformer.h++*

```

1  /*
2  This transformer implements constant-time transformations on a fat
3  heap.
4
5  Author: Jyrki Katajainen © 2010
6  */
7
8  #ifndef __CPHSTL_FAT_HEAP_TRANSFORMER__
9  #define __CPHSTL_FAT_HEAP_TRANSFORMER__
10
11 #include <algorithm> // std::swap
12 #include "comparator-proxy.h++"
13 #include <iostream>
14
15 namespace cphstl {
16     template <typename C, typename N>
17     class fat_heap_transformer {
18     public:
19
20         typedef C comparator_type;
21         typedef N node_type;
22
23     protected:
24
25         comparator_proxy<C> comparator;
26
27     public:
28
29         explicit fat_heap_transformer(C const & c = C())
30             : comparator(c) {
31         }
32
33         ~fat_heap_transformer() {
34         }

```

```

35
36 template <typename T, typename V>
37 N* reduce(N* p, N* q, T& tree_inventory, V& violation_inventory) {
38     N* a = cleaning_transformation(p, q);
39     N* b = cleaning_transformation(q, p);
40     if (a != b) {
41         a = parent_transformation(a, b, p, q);
42     }
43     N* d = pair_transformation(a, tree_inventory, violation_inventory);
44     return d;
45 }
46
47 private:
48
49 N* cleaning_transformation(N* q, N* fellow) {
50     N* papa = (*q).parent();
51     if ((*papa).right() == q) {
52         return papa;
53     }
54     N* r = (*q).parent();
55     papa = (*r).parent();
56     if ((*papa).right() == r && r == fellow) {
57         return papa;
58     }
59     if ((*papa).right() == r) { // q <-> r
60         N* p = (*q).left();
61         N* s = (*r).parent();
62         if (p != 0) {
63             (*p).parent(r);
64         }
65         (*r).left(p);
66         (*r).parent(q);
67         (*q).left(r);
68         (*q).parent(s);
69         (*s).right(q);
70         return s;
71     }
72     N* s = r;
73     if ((*q).rank() == (*s).rank()) {
74         s = (*s).parent();
75     }
76     if ((*s).is_marked()) {
77         s = (*s).parent();
78     }
79     N* v = (*s).right();
80     N* u = (*v).left();
81     N* t = (*u).left();
82     N* p = (*q).left();
83     if (v == fellow) { // q <-> u
84         if (p != 0) {
85             (*p).parent(u);
86         }
87         (*u).left(p);
88         (*u).parent(r);
89         (*r).left(u);
90         if (t != 0) {
91             (*t).parent(q);
92         }
93         (*q).left(t);
94         (*q).parent(v);
95         (*v).left(q);
96         return s;
97     }
98     if (p != 0) {
99         (*p).parent(v);

```

```

100     }
101     (*v).left(p);
102     (*v).parent(r);
103     (*r).left(v);
104     (*s).right(q);
105     (*q).parent(s);
106     (*q).left(u);
107     (*u).parent(q);
108     return s;
109 }
110
111 N* parent_transformation(N* a, N* b, N* r, N* u) {
112     if (comparator((*a).element(), (*b).element())) {
113         std::swap(b, a);
114         std::swap(u, r);
115     }
116     N* q = (*r).left();
117     N* t = (*u).left();
118     N* s = (*t).left();
119     (*q).parent(t); // r <-> t
120     (*t).left(q);
121     (*t).parent(a);
122     (*a).right(t);
123     if (s != 0) {
124         (*s).parent(r);
125     }
126     (*r).left(s);
127     (*r).parent(u);
128     (*u).left(r);
129     return b;
130 }
131
132 template <typename T, typename V>
133 N* pair_transformation(N* d, T& tree_inventory, V& violation_inventory) {
134     N* r = (*d).right();
135     N* q = (*r).left();
136     N* p = (*q).left();
137     N* c = (*d).left();
138     N* u = (*d).parent();
139     bool at_root = (*d).is_root();
140     bool on_right = true;
141     if (at_root) {
142         tree_inventory.extract(d);
143         (*d).is_root(false);
144     }
145     else if ((*u).right() != d) {
146         on_right = false;
147     }
148     violation_inventory.extract(d);
149     (*d).right(p);
150     if (p != 0) {
151         (*p).parent(d);
152     }
153     (*d).left(0);
154     (*d).parent(q);
155     (*d).rank((*d).rank() - 1);
156     (*q).left(d);
157     (*r).parent(0);
158     d = (*d).join(comparator, tree_inventory, violation_inventory);
159     if (at_root) {
160         (*d).is_root(true);
161         tree_inventory.insert(d);
162     }
163     else {
164         if (c != 0) {

```

```

165         (*c).parent(d);
166     }
167     (*d).left(c);
168     if (on_right) {
169         (*u).right(d);
170         (*d).parent(u);
171     }
172     else {
173         (*u).left(d);
174         (*d).parent(u);
175     }
176 }
177 return d;
178 }
179 };
180 }
181
182 #endif

```

Bit hacks

§ 7 *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 #include "bit-manipulation.h++"
12 #include <climits> // CHAR_BIT
13 #include <cstddef>
14 #include <iostream>
15
16 namespace cphstl {
17     template <typename W>
18     class bit_store {
19     public:
20
21         typedef W word_type;
22         typedef std::size_t size_type;
23
24         enum {word_size = CHAR_BIT * sizeof(word_type)};
25
26         explicit bit_store(word_type value = 0)
27             : word(value) {
28         }
29
30         operator word_type() const {
31             return word;
32         }
33
34         bool empty() const {
35             return word == 0;
36         }
37
38         size_type size() const {
39             return population_count(word);
40         }
41

```

```

42     size_type capacity() const {
43         return word_size;
44     };
45
46     template <typename I>
47     void set(I index) {
48         word_type mask = word_type(1) << word_type(index);
49         word &= ~ mask;
50         word |= mask;
51     }
52
53     template <typename I>
54     void set(I i, I j) { // set all bits whose index is in [i,j]
55         word_type mask = word_type(0);
56         if (word_type(j - i) < word_size) {
57             mask = word_type(1) << word_type(j - i);
58         }
59         mask -= 1;
60         mask <<= word_type(i);
61         word &= ~ mask;
62         word |= mask;
63     }
64
65     template <typename I>
66     void unset(I index) {
67         word_type mask = word_type(1) << word_type(index);
68         word &= ~ mask;
69     }
70
71     template <typename I>
72     void unset(I i, I j) {
73         word_type mask = word_type(0);
74         if (word_type(j - i) < word_size) {
75             mask = word_type(1) << word_type(j - i);
76         }
77         mask -= 1;
78         mask <<= word_type(i);
79         word &= ~ mask;
80     }
81
82     template <typename I>
83     bool get(I index) const {
84         word_type v = word_type(1) << index;
85         v = word & v;
86         return (v > 0);
87     }
88
89     size_type least_significant_one() const {
90         return trailing_zeros(word);
91     }
92
93     size_type most_significant_one() const {
94         return capacity() - leading_zeros(word) - 1;
95     }
96
97     size_type choose() const {
98         return most_significant_one();
99     }
100
101     bit_store operator|(bit_store const& other) const {
102         bit_store result = *this;
103         result.word |= other.word;
104         return result;
105     }
106

```

```

107     bit_store operator&(bit_store const& other) const {
108         bit_store result = *this;
109         result.word &= other.word;
110         return result;
111     }
112
113     bit_store & operator=(bit_store const& other) {
114         word = other.word;
115         return *this;
116     }
117
118     protected:
119
120         word_type word;
121     };
122 }
123
124 #endif

```

§ 8 *bit-manipulation.h++*

```

1 /*
2  Description: Machine-dependant bit manipulation in terms of Intel
3  History: Imported from Vuillemin's priority queue
4  Warning: Cross-platform code
5  Authors: Asger Bruun, Jyrki Katajainen © 2009, 2010
6 */
7
8 #ifndef __CPHSTL_BIT_MANIPULATION__
9 #define __CPHSTL_BIT_MANIPULATION__
10 #include <climits> // CHAR_BIT
11 #include <cstdint>
12
13 namespace cphstl {
14     template<typename T>
15     std::size_t leading_zeros(T n) {
16         T b(0);
17         while (n != 0) {
18             n >>= 1;
19             ++b;
20         }
21         return sizeof(T) * CHAR_BIT - b;
22     }
23
24     template<typename T>
25     std::size_t trailing_zeros(T n) {
26         T b(0);
27         while ((n & 1) == 0) {
28             n >>= 1;
29             ++b;
30         }
31         return b;
32     }
33
34     template<typename T>
35     std::size_t population_count(T n) {
36         T b(0);
37         while (n != 0) {
38             b += (n & 1);
39             n >>= 1;
40         }
41         return b;
42     }
43 }

```

```

44
45 #ifdef _MSC_VER
46 #include <intrin.h> // _BitScanForward
47 #include <cstdlib>
48
49 namespace cphstl {
50 #ifdef _M_X86
51 #pragma intrinsic(_bittest, _BitScanForward, _BitScanReverse, __popcnt)
52
53 bool bit_test(long const& n, long pos) {
54     return bool(_bittest((long *) &n, pos));
55 }
56
57 std::size_t leading_zeros(unsigned long n) {
58     unsigned long index;
59     (void) _BitScanForward(&index, n);
60     return sizeof(std::size_t) * CHAR_BIT - index - 1;
61 };
62
63 std::size_t trailing_zeros(unsigned long n) {
64     unsigned long index;
65     (void) _BitScanReverse(&index, n);
66     return index;
67 };
68
69 #endif
70
71 #ifdef _M_X64
72 #pragma intrinsic(_BitScanForward, _BitScanReverse, __popcnt)
73
74 bool bit_test(std::size_t const& n, std::size_t pos) {
75     return bool(_bittest64((std::__int64 *) &n, pos));
76 }
77
78 std::size_t leading_zeros(unsigned std::__int64 n) {
79     unsigned long index;
80     (void) _BitScanForward(&index, n);
81     return sizeof(std::size_t) * CHAR_BIT - index - 1;
82 };
83
84 std::size_t trailing_zeros(unsigned std::__int64 n) {
85     unsigned long index;
86     (void) _BitScanReverse64(&index, n);
87     return index;
88 };
89
90 std::size_t inline population_count(unsigned std::__int64 n) {
91     return __popcnt64(n);
92 };
93
94 #endif
95 }
96
97 #endif
98
99 #ifdef __GNUC__
100
101 namespace cphstl {
102 // Ref: gcc.gnu.org/onlinedocs/gcc-4.4.0/gcc/Other-Builtins.html#index-
103 // t_005f_005fbuiltin_005fffs-2894
104
105 std::size_t leading_zeros(unsigned long long n) {
106     return __builtin_clzll(n);
107 }

```

```

108
109 std::size_t trailing_zeros(unsigned long long n) {
110     return __builtin_ctzll(n);
111 }
112
113 std::size_t population_count(unsigned long long n) {
114     return __builtin_popcountll(n);
115 }
116
117 std::size_t leading_zeros(unsigned long n) {
118     return __builtin_clzl(n);
119 }
120
121 std::size_t trailing_zeros(unsigned long n) {
122     return __builtin_ctzl(n);
123 }
124
125 std::size_t population_count(unsigned long n) {
126     return __builtin_popcountl(n);
127 }
128
129 std::size_t leading_zeros(unsigned int n) {
130     return __builtin_clz(n);
131 }
132
133 std::size_t trailing_zeros(unsigned int n) {
134     return __builtin_ctz(n);
135 }
136
137 std::size_t population_count(unsigned int n) {
138     return __builtin_popcount(n);
139 }
140 }
141 #endif
142 #endif
143 #endif

```

STL interface

§ 9 *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 <cstdlib> // std::size_t and std::ptrdiff_t
38
39  namespace cphstl {
40  template <typename V, typename C, typename A, typename E, typename R,
41           typename I, typename J>
42  class meldable_priority_queue {
43  public:
44
45      // types
46
47      typedef V value_type;
48      typedef C comparator_type;
49      typedef A allocator_type;
50      typedef E encapsulator_type;
51      typedef R realizator_type;
52      typedef I iterator;
53      typedef J const_iterator;
54      typedef typename R::reference reference;
55      typedef typename R::const_reference const_reference;
56      typedef V* pointer;
57      typedef V const* const_pointer;
58      typedef std::size_t size_type;
59      typedef std::ptrdiff_t difference_type;
60      typedef std::reverse_iterator<iterator> reverse_iterator;
61      typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
62      typedef meldable_priority_queue<V, C, A, E, R, I, J> container_type;
63
64  protected:
65
66      typedef typename A::template rebind<E>::other encapsulator_allocator_type;
67
68      R realizator;
69      encapsulator_allocator_type allocator;
70
71      E* create(V const &);
72      void destroy(E*);
73
74      template <typename K>
75      void insert(K, K);
76
77  public:
78
79      // structors
80
81      explicit meldable_priority_queue(C const & = C(), A const & = A());
82      meldable_priority_queue(meldable_priority_queue const &);
83      meldable_priority_queue & operator=(meldable_priority_queue const &);
84      ~meldable_priority_queue();
85
86      // iterators
87
88      iterator begin();

```

```

89     const_iterator begin() const;
90     iterator end();
91     const_iterator end() const;
92
93     // accessors
94
95     allocator_type get_allocator() const;
96     comparator_type get_comparator() const;
97     bool empty() const;
98     size_type size() const;
99     size_type max_size() const;
100    const_iterator top() const;
101
102    // modifiers
103
104    iterator top();
105    iterator push(V const &);
106    void pop();
107    void erase(iterator);
108    void increase(iterator, V const &);
109    void clear();
110    void meld(meldable_priority_queue &);
111    void swap(meldable_priority_queue &);
112
113 };
114
115 // algorithms
116
117 template <typename V, typename C, typename A, typename E,
118          typename R, typename I, typename J>
119 meldable_priority_queue<V, C, A, E, R, I, J>&
120 meld(meldable_priority_queue<V, C, A, E, R, I, J>&,
121      meldable_priority_queue<V, C, A, E, R, I, J>&);
122
123 template <typename V, typename C, typename A, typename E,
124          typename R, typename I, typename J>
125 void swap(meldable_priority_queue<V, C, A, E, R, I, J>&,
126          meldable_priority_queue<V, C, A, E, R, I, J>&);
127 }
128
129 #include "meldable-priority-queue.i++"
130 #endif

```

§ 10 *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      template <typename V, typename C, typename A, typename E,
10             typename R, typename I, typename J>
11      E*
12      meldable_priority_queue<V, C, A, E, R, I, J>::create(V const & v) {
13          E* p = allocator.allocate(1);
14          try {
15              new (p) E(v, allocator);
16          }
17          catch (...) {
18              destroy(p);
19              throw;

```

```

20     }
21     return p;
22 }
23
24 template <typename V, typename C, typename A, typename E,
25           typename R, typename I, typename J>
26 void
27 meldable_priority_queue<V, C, A, E, R, I, J>::destroy(E* p) {
28     p->~E();
29     allocator.deallocate(p, 1);
30 }
31
32 template <typename V, typename C, typename A, typename E,
33           typename R, typename I, typename J>
34 template <typename K>
35 void
36 meldable_priority_queue<V, C, A, E, R, I, J>::insert(K b, K e) {
37     meldable_priority_queue q;
38     E* d;
39     try {
40         for (K c = b; c != e; ++c) {
41             d = create(*c);
42             I t(d);
43             (void) q.realizator.insert(t);
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(allocator) {
59 }
60
61 template <typename V, typename C, typename A, typename E,
62           typename R, typename I, typename J>
63 meldable_priority_queue<V, C, A, E, R, I, J>::meldable_priority_queue(
64     meldable_priority_queue const& other) {
65     meldable_priority_queue q;
66     try {
67         J first = other.begin();
68         J past_the_end = other.end();
69         q.insert(first, past_the_end);
70     }
71     catch (...) {
72         while (q.size() != 0) {
73             I t = q.realizator.extract();
74             destroy(t);
75         }
76         throw;
77     }
78     (*this).swap(q);
79 }
80
81 template <typename V, typename C, typename A, typename E,
82           typename R, typename I, typename J>
83 typename meldable_priority_queue<V, C, A, E, R, I, J>::container_type &
84 meldable_priority_queue<V, C, A, E, R, I, J>::operator=(

```

```

85         meldable_priority_queue const& other) {
86     meldable_priority_queue q;
87     try {
88         J first = other.begin();
89         J past_the_end = other.end();
90         q.insert(first, past_the_end);
91     }
92     catch (...) {
93         while (q.size() != 0) {
94             I t = q.realizator.extract();
95             destroy(t);
96         }
97         throw;
98     }
99     (*this).swap(q);
100    while (q.size() != 0) {
101        I t = q.realizator.extract();
102        destroy(t);
103    }
104    return *this;
105 }
106
107 template <typename V, typename C, typename A, typename E,
108         typename R, typename I, typename J>
109 meldable_priority_queue<V, C, A, E, R, I, J>::~meldable_priority_queue() {
110     clear();
111 }
112
113 template <typename V, typename C, typename A, typename E,
114         typename R, typename I, typename J>
115 A
116 meldable_priority_queue<V, C, A, E, R, I, J>::get_allocator() const {
117     return A(allocator);
118 }
119
120 template <typename V, typename C, typename A, typename E,
121         typename R, typename I, typename J>
122 C
123 meldable_priority_queue<V, C, A, E, R, I, J>::get_comparator() const {
124     return realizator.get_comparator();
125 }
126
127 template <typename V, typename C, typename A, typename E,
128         typename R, typename I, typename J>
129 I
130 meldable_priority_queue<V, C, A, E, R, I, J>::begin() {
131     return I(realizator.begin());
132 }
133
134 template <typename V, typename C, typename A, typename E,
135         typename R, typename I, typename J>
136 J
137 meldable_priority_queue<V, C, A, E, R, I, J>::begin() const {
138     return J(realizator.begin());
139 }
140
141 template <typename V, typename C, typename A, typename E,
142         typename R, typename I, typename J>
143 I
144 meldable_priority_queue<V, C, A, E, R, I, J>::end() {
145     return I(realizator.end());
146 }
147
148 template <typename V, typename C, typename A, typename E,
149         typename R, typename I, typename J>

```

```

150     J
151     meldable_priority_queue<V, C, A, E, R, I, J>::end() const {
152         return J(realizator.end());
153     }
154
155     template <typename V, typename C, typename A, typename E,
156             typename R, typename I, typename J>
157     bool
158     meldable_priority_queue<V, C, A, E, R, I, J>::empty() const {
159         return (*this).size() == size_type(0);
160     }
161
162     template <typename V, typename C, typename A, typename E,
163             typename R, typename I, typename J>
164     typename meldable_priority_queue<V, C, A, E, R, I, J>::size_type
165     meldable_priority_queue<V, C, A, E, R, I, J>::size() const {
166         return realizator.size();
167     }
168
169     template <typename V, typename C, typename A, typename E,
170             typename R, typename I, typename J>
171     typename meldable_priority_queue<V, C, A, E, R, I, J>::size_type
172     meldable_priority_queue<V, C, A, E, R, I, J>::max_size() const {
173         return realizator.max_size();
174     }
175
176     template <typename V, typename C, typename A, typename E,
177             typename R, typename I, typename J>
178     J
179     meldable_priority_queue<V, C, A, E, R, I, J>::top() const {
180         return J(realizator.top());
181     }
182
183     template <typename V, typename C, typename A, typename E,
184             typename R, typename I, typename J>
185     I
186     meldable_priority_queue<V, C, A, E, R, I, J>::top() {
187         return I(realizator.top());
188     }
189
190     template <typename V, typename C, typename A, typename E,
191             typename R, typename I, typename J>
192     I
193     meldable_priority_queue<V, C, A, E, R, I, J>::push(V const& v) {
194         E* p = create(v);
195         I t(p);
196         t = realizator.insert(t);
197         return t;
198     }
199
200     template <typename V, typename C, typename A, typename E,
201             typename R, typename I, typename J>
202     void
203     meldable_priority_queue<V, C, A, E, R, I, J>::pop() {
204         I t = realizator.top();
205         (void) realizator.extract(t);
206         destroy(t);
207     }
208
209     template <typename V, typename C, typename A, typename E,
210             typename R, typename I, typename J>
211     void
212     meldable_priority_queue<V, C, A, E, R, I, J>::erase(I t) {
213         (void) realizator.extract(t);
214         destroy(t);

```

```

215 }
216
217 template <typename V, typename C, typename A, typename E,
218          typename R, typename I, typename J>
219 void
220 meldable_priority_queue<V, C, A, E, R, I, J>::increase(I t, V const & v) {
221     realizator.increase(t, v);
222 }
223
224 template <typename V, typename C, typename A, typename E,
225          typename R, typename I, typename J>
226 void
227 meldable_priority_queue<V, C, A, E, R, I, J>::clear() {
228     while (realizator.size() != 0) {
229         I t = realizator.extract();
230         destroy(t);
231     }
232 }
233
234 template <typename V, typename C, typename A, typename E,
235          typename R, typename I, typename J>
236 void
237 meldable_priority_queue<V, C, A, E, R, I, J>::meld(meldable_priority_queue & q) {
238     realizator.meld(q.realizator);
239 }
240
241 template <typename V, typename C, typename A, typename E,
242          typename R, typename I, typename J>
243 void
244 meldable_priority_queue<V, C, A, E, R, I, J>::swap(meldable_priority_queue & q) {
245     realizator.swap(q.realizator);
246 }
247
248 template <typename V, typename C, typename A, typename E,
249          typename R, typename I, typename J>
250 meldable_priority_queue<V, C, A, E, R, I, J>&
251 meld(meldable_priority_queue<V, C, A, E, R, I, J>& r,
252      meldable_priority_queue<V, C, A, E, R, I, J>& s) {
253     r.meld(s.realizator);
254     return r;
255 }
256
257 template <typename V, typename C, typename A, typename E,
258          typename R, typename I, typename J>
259 void swap(meldable_priority_queue<V, C, A, E, R, I, J>& r,
260          meldable_priority_queue<V, C, A, E, R, I, J>& s) {
261     r.swap(s.realizator);
262 }
263 }

```

STL iterator

§ 11 *node-iterator.h++*

```

1 /*
2  The idea of combining iterators and const iterators into the same
3  class is taken from [Matt Austern. Defining iterators and const
4  iterators. C/C++ User's Journal 19,1 (2001), 74-79].
5
6  Authors: Jyrki Katajainen, Bo Simonsen © 2006, 2008
7 */
8
9 #ifndef __CPHSTL_NODE_ITERATOR__
10 #define __CPHSTL_NODE_ITERATOR__

```

```

11
12 #include <cstddef> // std::ptrdiff_t
13 #include <iostream> // std::ostream
14 #include <iterator> // std::bidirectional_iterator_tag
15 #include <string> // std::string
16
17 namespace leda {
18     template < typename K, typename I, typename C, typename R >
19     class dictionary;
20 }
21
22 namespace {
23     // if statement for compile-time meta-programming
24
25     template < bool, typename T, typename U >
26     class if_then_else;
27
28     template < typename T, typename U >
29     class if_then_else<true, T, U> {
30     public:
31         typedef T type;
32     };
33
34     template < typename T, typename U >
35     class if_then_else<false, T, U> {
36     public:
37         typedef U type;
38     };
39 }
40
41 namespace cphstl {
42     /* Forward declarations of container classes */
43
44     template < typename V, typename A, typename R, typename I, typename J >
45     class list;
46
47     template < typename V, typename C, typename A, typename R,
48             typename I, typename J >
49     class multiset;
50
51     template < typename V, typename C, typename A, typename R,
52             typename I, typename J >
53     class set;
54
55     template < typename K, typename V, typename C, typename A, typename R,
56             typename I, typename J >
57     class set_base;
58
59     template < typename K, typename V, typename C, typename A, typename R,
60             typename I, typename J >
61     class map;
62
63     template < typename K, typename V, typename C, typename A, typename R,
64             typename I, typename J >
65     class map_base;
66
67     template < typename V, typename C, typename A, typename E,
68             typename R, typename I, typename J >
69     class meldable_priority_queue;
70
71     template < typename N, bool is_const = false >
72     class node_iterator {
73     public:
74
75         // types

```

```

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

```

```

138     node_iterator & operator=(node_iterator const &);
139     ~node_iterator();
140
141     // operators
142
143     reference operator*() const;
144     pointer operator->() const;
145     pointer operator->();
146     node_iterator & operator++();
147     node_iterator operator++(int);
148     node_iterator & operator--();
149     node_iterator operator--(int);
150
151     template <bool both>
152     bool operator==(node_iterator<N, both> const &) const;
153
154     template <bool both>
155     bool operator!=(node_iterator<N, both> const &) const;
156
157     private:
158
159     // converters to be used by the friends
160
161     node_iterator(node_pointer); // node_pointer --> iterator
162     operator node_pointer() const; // iterator --> node_pointer
163     operator std::string() const; // iterator --> string
164
165     // variables
166
167     node_pointer link;
168 };
169 }
170
171 #include "node-iterator.i++"
172 #endif

```

§ 12 *node-iterator.i++*

```

1 /*
2  * Implementation of cphstl::node_iterator
3  *
4  * Author: Jyrki Katajainen, Bo Simonsen, April 2008
5  *
6  * General design idea is proposed by J. Katajainen and B. Simonsen
7  */
8
9 #include <sstream> // defines std::stringstream
10
11 namespace cphstl {
12     // default constructor
13
14     template <typename N, bool is_const>
15     node_iterator<N, is_const>::node_iterator()
16     : link(0) {
17     }
18
19     // copy constructor
20
21     template <typename N, bool is_const>
22     node_iterator<N, is_const>::node_iterator(node_iterator<N, false> const & a)
23     : link(a.link) {
24     }
25     template <typename N, bool is_const>
26     template <typename R>

```

```

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

```

```

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

```

```

156
157 template <typename N, bool both>
158 std::ostream &
159 operator<<(std::ostream & s, node_iterator<N, both> const & i) {
160     s << std::string(i);
161     return s;
162 }
163
164 }

```

Proxy

§ 13 *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     template <typename C>
14     class comparator_proxy {
15     public:
16
17         comparator_proxy(C const & c = C()) {
18             (*this).c = new C(c);
19         }
20
21         comparator_proxy(comparator_proxy const & cp) {
22             (*this).c = new C(*cp.c);
23         }
24
25         comparator_proxy operator=(comparator_proxy const & cp) {
26             delete (*this).c;
27             (*this).c = new C(*cp.c);
28             return (*this);
29         }
30
31         comparator_proxy operator=(C const & c) {
32             delete (*this).c;
33             (*this).c = new C(c);
34             return (*this);
35         }
36
37         ~comparator_proxy() {
38             delete (*this).c;
39         }
40
41         template <typename T, typename U>
42         bool operator()(T const & t, U const & u) const {
43             return (*c).operator()(t, u);
44         }
45
46         C subject() const {
47             return *c;
48         }
49
50     private:

```

```

51
52     C* c;
53 };
54 }
55
56 #endif

```

Decorator

§ 14 *top-decorator.h++*

```

1  /*
2   This data-structural transformation converts a priority-queue
3   realizator supporting find-min in  $F(n)$  time, insert in  $I(n)$  time,
4   decrease in  $D(n)$ , and extract in  $E(n)$  time to a priority queue that
5   supports find-min in  $O(1)$  time, insert in  $I(n) + O(1)$  time, decrease
6   in  $D(n) + O(1)$  time, and extract-min in  $F(n) + E(n) + O(1)$  time. The
7   idea is simply to keep a pointer to the node storing the minimum.
8
9   Authors: Asger Bruun, Jyrki Katajainen © 2009, 2010
10 */
11
12 #ifndef __CPHSTL_TOP_DECORATOR__
13 #define __CPHSTL_TOP_DECORATOR__
14
15 #include <algorithm> // std::swap
16 #include <iostream>
17
18 namespace cphstl {
19     template <typename R>
20     class top_decorator :
21     public R {
22     public:
23
24         // types
25
26         typedef typename R::element_type value_type;
27         typedef typename R::comparator_type comparator_type;
28         typedef typename R::allocator_type allocator_type;
29         typedef typename R::locator_type locator_type;
30         typedef typename R::size_type size_type;
31
32     private:
33
34         // variables
35
36         locator_type top_;
37
38     public:
39
40         // structors
41
42         top_decorator(comparator_type const& c = comparator_type(),
43                     allocator_type const& a = allocator_type())
44             : R(c, a), top_(0) {
45         }
46
47         // accessors
48
49         locator_type top() const {
50             return top_;
51         }
52
53         // modifiers

```

```

54
55 locator_type insert(locator_type p) {
56     locator_type q = R::insert(p);
57     comparator_type comparator = R::get_comparator();
58     if (top_ == 0 || comparator((*top_).element(), (*q).element())) {
59         top_ = p;
60     }
61     return q;
62 }
63
64 locator_type extract() {
65     locator_type q(R::extract());
66     if (q != top_) {
67         return q;
68     }
69     if (q == top_ && R::size() == 0) {
70         top_ = 0;
71         return q;
72     }
73     locator_type p(R::extract());
74     (void) R::insert(q);
75     return p;
76 }
77
78 locator_type extract(locator_type q) {
79     locator_type p(R::extract(q));
80     if (q == top_) {
81         top_ = R::top();
82     }
83     return p;
84 }
85
86 void increase(locator_type p, value_type const& v) {
87     R::increase(p, v);
88     comparator_type comparator = R::get_comparator();
89     if (comparator((*top_).element(), (*p).element())) {
90         top_ = p;
91     }
92 }
93
94 void meld(top_decorator & other) {
95     R::meld(static_cast<R>(other));
96     comparator_type comparator = R::get_comparator();
97     if (top_ == 0) {
98         top_ = other.top_;
99     }
100     else if (other.top_ == 0) {
101         top_ = top_;
102     }
103     else if (comparator((*top_).element(), (*other.top_).element())) {
104         top_ = other.top_;
105     }
106     else {
107         top_ = top_;
108     }
109     other.top_ = 0;
110 }
111
112 void swap(top_decorator & other) {
113     R::swap(static_cast<R>(other));
114     std::swap(top_, other.top_);
115 }
116
117 };
118 }

```

```
119 #endif
```

Benchmark aids

§ 15 *fat-heap.i++*

```
1 #include "arena-based-violation-inventory.h++"
2 #include "fat-heap-node.h++"
3 #include "fat-heap-transformer.h++"
4 #include "meldable-priority-queue.h++"
5 #include <memory>
6 #include "node-iterator.h++"
7 #include "relaxed-heap.h++"
8 #include "semi-sorted-tree-inventory.h++"
9
10 long long comps = 0;
11
12 template <typename T>
13 class counting_comparator {
14 public:
15
16     bool operator()(T const & a, T const & b) const {
17         ++comps;
18         return a < b;
19     }
20 };
21
22 typedef long long E;
23 typedef counting_comparator<E> C;
24
25 typedef cphstl::fat_heap_node<E> N;
26 typedef cphstl::semi_sorted_tree_inventory<3, C, N> T;
27 typedef cphstl::fat_heap_transformer<C, N> S;
28 typedef cphstl::arena_based_violation_inventory<C, N, S> V;
29 typedef cphstl::relaxed_heap<E, C, N, T, V> R;
30
31 typedef cphstl::node_iterator<N> I;
32 typedef cphstl::node_iterator<N, true> J;
33 typedef std::allocator<E> A;
34 typedef cphstl::meldable_priority_queue<E, C, A, N, R, I, J> Q;
```

§ 16 *slow-increase-fat-heap.i++*

```
1 #include "blank-violation-inventory.h++"
2 #include "fat-heap-node.h++"
3 #include "meldable-priority-queue.h++"
4 #include <memory>
5 #include "node-iterator.h++"
6 #include "relaxed-heap.h++"
7 #include "semi-sorted-tree-inventory.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 E;
22 typedef counting_comparator<E> C;
23 typedef std::allocator<E> A;
24
25 typedef cphstl::fat_heap_node<E, A> N;
26 typedef cphstl::semi_sorted_tree_inventory<3, C, A, N> T;
27 typedef cphstl::blank_violation_inventory<C, A, N> V;
28 typedef cphstl::relaxed_heap<E, C, A, N, T, V> R;
29
30 typedef cphstl::node_iterator<N> I;
31 typedef cphstl::node_iterator<N, true> J;
32 typedef cphstl::meldable_priority_queue<E, C, A, N, R, I, J> Q;

```

§ 17 *fast-top-fat-heap.i++*

```

1 #include "arena-based-violation-inventory.h++"
2 #include "fat-heap-node.h++"
3 #include "fat-heap-transformer.h++"
4 #include "meldable-priority-queue.h++"
5 #include <memory>
6 #include "priority-queue-iterator.h++"
7 #include "relaxed-heap.h++"
8 #include "semi-sorted-tree-inventory.h++"
9 #include "top-decorator.h++"
10
11 long long comps = 0;
12
13 template <typename T>
14 class counting_comparator {
15 public:
16
17     bool operator()(T const & a, T const & b) const {
18         ++comps;
19         return a < b;
20     }
21 };
22
23 typedef long long E;
24 typedef counting_comparator<E> C;
25 typedef std::allocator<E> A;
26
27 typedef cphstl::fat_heap_node<E, A> N;
28 typedef cphstl::semi_sorted_tree_inventory<3, C, A, N> T;
29 typedef cphstl::fat_heap_transformer<C, N> S;
30 typedef cphstl::arena_based_violation_inventory<C, A, N, S> V;
31 typedef cphstl::relaxed_heap<E, C, A, N, T, V> R;
32 typedef cphstl::top_decorator<R> D;
33
34 // Warning: iterators do not work because of the two levels of nodes
35
36 typedef cphstl::priority_queue_iterator<N, D> I;
37 typedef cphstl::priority_queue_iterator<N, D, true> J;
38 typedef cphstl::meldable_priority_queue<E, C, A, N, D, I, J> Q;

```

Benchmark drivers

§ 18 *push-time.c++*

```

1 #include <algorithm>
2 #include <cstdlib> // std::rand, std::srand
3 #include <ctime>
4 #include <iostream>
5

```

```

6 #ifndef NUMBER
7 #define NUMBER 1000000
8 #endif
9
10 #ifndef DATA
11 #define DATA increasing_sequence
12 #endif
13
14 #include "data-structure.i++" // Q comes from here
15 #include <iterator> // defines std::iterator_traits
16
17 template <typename I>
18 void increasing_sequence(I first, I past_the_end) {
19     typedef typename std::iterator_traits<I>::value_type V;
20     I q;
21     for (q = first; q != past_the_end; ++q) {
22         *q = V(unsigned(q - first));
23     }
24 }
25
26 template <typename I>
27 void decreasing_sequence(I first, I past_the_end) {
28     typedef typename std::iterator_traits<I>::value_type V;
29     I q;
30     for (q = first; q != past_the_end; ++q) {
31         *q = V(unsigned(past_the_end - q));
32     }
33 }
34
35 template <typename I>
36 void random_sequence(I first, I past_the_end) {
37     typedef typename std::iterator_traits<I>::value_type V;
38     I q;
39     for (q = first; q != past_the_end; ++q) {
40         *q = V(unsigned(std::rand()));
41     }
42 }
43
44 template <typename I>
45 void random_sequence(I first, I past_the_end, unsigned long seed) {
46     std::srand(seed);
47     typedef typename std::iterator_traits<I>::value_type V;
48     I q;
49     for (q = first; q != past_the_end; ++q) {
50         *q = V(unsigned(std::rand()));
51     }
52 }
53
54 int main() {
55     typedef Q::value_type E;
56
57     int const number_of_elements = NUMBER;
58     int repetitions = 10000000 / number_of_elements;
59     if (repetitions < 3) {
60         repetitions = 3;
61     }
62
63     E* a = new E[number_of_elements];
64     DATA(&a[0], &a[number_of_elements]);
65
66     Q* many = new Q[repetitions];
67     std::clock_t start = std::clock();
68     for (int k = 0; k != repetitions; ++k) {
69         for (int i = 0; i != number_of_elements; ++i) {
70             (void) many[k].push(a[i]);

```

```

71     }
72 }
73 std::clock_t stop = std::clock();
74 for (int k = 0; k != repetitions; ++k) {
75     many[k].clear();
76 }
77 delete[] many;
78 delete[] a;
79
80 double running_time = double(stop - start)/double(CLOCKS_PER_SEC);
81 double time_per_run = 1000000.0 * running_time / double(repetitions);
82 double time_per_operation = time_per_run / double(number_of_elements);
83 std::cout << number_of_elements << " " << time_per_operation << "\n";
84
85 return 0;
86 }

```

§ 19 *increase-time.cpp*

```

1 #include <algorithm>
2 #include <cstdlib>
3 #include <ctime>
4 #include <iostream>
5 #include <vector>
6
7 #ifndef NUMBER
8 #define NUMBER 1000000
9 #endif
10
11 #include "data-structure.i++" // Q comes from here
12
13 int main() {
14     typedef Q::value_type E;
15
16     int const n = NUMBER;
17     int repetitions = 1000000 / n;
18     if (repetitions < 3) {
19         repetitions = 3;
20     }
21
22     E* a = new E[n];
23     for (int i = 0; i < n; i++) {
24         a[i] = (E) i;
25     }
26
27     srand(1);
28     for (int i = 0; i < n; i++) {
29         std::swap(a[i], a[rand() % (n)]);
30     }
31
32     std::vector<Q::iterator> v(n);
33     Q* many = new Q[repetitions];
34     std::clock_t start = std::clock();
35     for (int k = 0; k != repetitions; ++k) {
36         for (int i = 0; i != n; ++i) {
37             Q::iterator p = many[k].push(a[i]);
38             v[a[i]] = p;
39         }
40         Q::iterator p = many[k].top();
41     }
42     std::clock_t stop = std::clock();
43     double dual_time = double(stop - start)/double(CLOCKS_PER_SEC);
44     for (int k = 0; k != repetitions; ++k) {
45         many[k].clear();

```

```

46 }
47
48 start = std::clock();
49 for (int k = 0; k != repetitions; ++k) {
50     for (int i = 0; i != n; ++i) {
51         Q::iterator p = many[k].push(a[i]);
52         v[a[i]] = p;
53     }
54     Q::iterator p = many[k].top();
55     for (int i = 0; i != n; ++i) {
56         many[k].increase(v[i], i + n);
57     }
58 }
59 stop = std::clock();
60 for (int k = 0; k != repetitions; ++k) {
61     many[k].clear();
62 }
63 v.clear();
64 delete[] many;
65 delete[] a;
66
67 double running_time = double(stop - start)/double(CLOCKS_PER_SEC);
68 running_time -= dual_time;
69 double time_per_run = 1000000.0* running_time / double(repetitions);
70 double time_per_operation = time_per_run / double(n);
71 std::cout << n << " " << time_per_operation << "\n";
72
73 return 0;
74 }

```

§ 20 *erase-time.cpp*

```

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

```

```

33 Q* many = new Q[repetitions];
34 std::clock_t start = std::clock();
35 for (int k = 0; k != repetitions; ++k) {
36     for (int i = 0; i != n; ++i) {
37         Q::iterator p = many[k].push(a[i]);
38         v[i] = p;
39     }
40     Q::iterator p = many[k].top();
41 }
42 std::clock_t stop = std::clock();
43 double dual_time = double(stop - start)/double(CLOCKS_PER_SEC);
44 for (int k = 0; k != repetitions; ++k) {
45     many[k].clear();
46 }
47
48 start = std::clock();
49 for (int k = 0; k != repetitions; ++k) {
50     for (int i = 0; i != n; ++i) {
51         Q::iterator p = many[k].push(a[i]);
52         v[i] = p;
53     }
54     Q::iterator p = many[k].top();
55     for (int i = 0; i != n; ++i) {
56         many[k].erase(v[i]);
57     }
58 }
59 stop = std::clock();
60 v.clear();
61 delete[] many;
62 delete[] a;
63
64 double running_time = double(stop - start)/double(CLOCKS_PER_SEC);
65 running_time -= dual_time;
66 double time_per_run = 1000000.0 * running_time / double(repetitions);
67 double time_per_operation = time_per_run / double(n);
68 std::cout << n << " " << time_per_operation << "\n";
69
70 return 0;
71 }

```

§ 21 *pop-time.cpp*

```

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

```

```

23
24 for (int i = 0; i < n; i++) {
25     a[i] = (E) i;
26 }
27
28 srand(1);
29 for (int i = 0; i < n; i++) {
30     std::swap(a[i], a[rand() % (n)]);
31 }
32
33 Q* many = new Q[repetitions];
34 std::clock_t start = std::clock();
35 for (int k = 0; k != repetitions; ++k) {
36     for (int i = 0; i != n; ++i) {
37         (void) many[k].push(a[i]);
38     }
39 }
40 std::clock_t stop = std::clock();
41 double dual_time = double(stop - start)/double(CLOCKS_PER_SEC);
42 for (int k = 0; k != repetitions; ++k) {
43     many[k].clear();
44 }
45
46 start = std::clock();
47 for (int k = 0; k != repetitions; ++k) {
48     for (int i = 0; i != n; ++i) {
49         (void) many[k].push(a[i]);
50     }
51     for (int i = 0; i != n; ++i) {
52         many[k].pop();
53     }
54 }
55 stop = std::clock();
56 delete[] many;
57 delete[] a;
58
59 double running_time = double(stop - start)/double(CLOCKS_PER_SEC);
60 running_time -= dual_time;
61 double time_per_run = 1000000.0 * running_time / double(repetitions);
62 double time_per_operation = time_per_run / double(n);
63 std::cout << n << " " << time_per_operation << "\n";
64
65 return 0;
66 }

```

§ 22 *push-comp.cpp*

```

1 #include <algorithm>
2 #include <cstdlib> // std::rand, std::srand
3 #include <ctime>
4 #include <iostream>
5
6 #ifndef NUMBER
7 #define NUMBER 1000000
8 #endif
9
10 #ifndef DATA
11 #define DATA increasing_sequence
12 #endif
13
14 #include "data-structure.i++" // Q comes from here
15 #include <iterator> // defines std::iterator_traits
16
17 template <typename I>

```

```

18 void increasing_sequence(I first, I past_the_end) {
19     typedef typename std::iterator_traits<I>::value_type V;
20     I q;
21     for (q = first; q != past_the_end; ++q) {
22         *q = V(unsigned(q - first));
23     }
24 }
25
26 template <typename I>
27 void decreasing_sequence(I first, I past_the_end) {
28     typedef typename std::iterator_traits<I>::value_type V;
29     I q;
30     for (q = first; q != past_the_end; ++q) {
31         *q = V(unsigned(past_the_end - q));
32     }
33 }
34
35 template <typename I>
36 void random_sequence(I first, I past_the_end) {
37     typedef typename std::iterator_traits<I>::value_type V;
38     I q;
39     for (q = first; q != past_the_end; ++q) {
40         *q = V(unsigned(std::rand()));
41     }
42 }
43
44 template <typename I>
45 void random_sequence(I first, I past_the_end, unsigned long seed) {
46     std::srand(seed);
47     typedef typename std::iterator_traits<I>::value_type V;
48     I q;
49     for (q = first; q != past_the_end; ++q) {
50         *q = V(unsigned(std::rand()));
51     }
52 }
53
54 int main() {
55     typedef Q::value_type E;
56
57     int const n = NUMBER;
58
59     E* a = new E[n];
60     auto total = comps;
61     Q q;
62
63     int repetitions = 1000000 / n;
64     if (repetitions < 3) {
65         repetitions = 3;
66     }
67
68     for (int k = 0; k < repetitions; ++k) {
69         DATA(&a[0], &a[n]);
70
71         comps = 0;
72         for (int i = 0; i != n; ++i) {
73             (void) q.push(a[i]);
74         }
75         total += comps;
76         q.clear();
77     }
78
79     double comp_count = double(total) / double(repetitions);
80     double comp_per_operation = comp_count / double(n);
81     std::cout << n << " " << comp_per_operation << "\n";
82

```

```

83 delete[] a;
84 return 0;
85 }

```

§ 23 *increase-comp.cpp*

```

1 #include <algorithm>
2 #include <cmath>
3 #include <cstdlib>
4 #include <iostream>
5 #include <vector>
6
7 #ifndef NUMBER
8 #define NUMBER 1000000
9 #endif
10
11 #include "data-structure.i++" // Q comes from here
12
13 int main() {
14     typedef Q::value_type E;
15
16     int const n = NUMBER;
17     E* a = new E[n];
18     srand(1);
19     auto total = comps;
20     Q q;
21     std::vector<Q::iterator> v(n);
22
23     int repetitions = 10000000 / n;
24     if (repetitions < 3) {
25         repetitions = 3;
26     }
27
28     for (int k = 0; k < repetitions; ++k) {
29         for (int i = 0; i < n; i++) {
30             a[i] = (E) i;
31         }
32         for (int i = 0; i < n; i++) {
33             std::swap(a[i], a[rand() % (n)]);
34         }
35         for (int i = 0; i != n; ++i) {
36             Q::iterator p = q.push(a[i]);
37             v[a[i]] = p;
38         }
39         Q::iterator p = q.top();
40
41         comps = 0;
42         for (int i = 0; i != n; ++i) {
43             q.increase(v[i], i + n);
44         }
45         total += comps;
46         q.clear();
47         v.clear();
48     }
49
50     double comp_count = double(total) / double(repetitions);
51     double comp_per_operation = comp_count / double(n);
52     // double factor = comp_per_operation / std::log2(double(n));
53     std::cout << n << " " << comp_per_operation << "\n";
54
55     delete[] a;
56     return 0;
57 }

```

§ 24 *erase-comp.cpp*

```

1 #include <algorithm>
2 #include <cmath>
3 #include <cstdlib>
4 #include <iostream>
5 #include <vector>
6
7 #ifndef NUMBER
8 #define NUMBER 1000000
9 #endif
10
11 #include "data-structure.i++" // Q comes from here
12
13 int main() {
14     typedef Q::value_type E;
15
16     int const n = NUMBER;
17     E* a = new E[n];
18     srand(1);
19     auto total = comps;
20     Q q;
21     std::vector<Q::iterator> v;
22
23     int repetitions = 10000000 / n;
24     if (repetitions < 3) {
25         repetitions = 3;
26     }
27
28     for (int k = 0; k < repetitions; ++k) {
29         for (int i = 0; i < n; i++) {
30             a[i] = (E) i;
31         }
32         for (int i = 0; i < n; i++) {
33             std::swap(a[i], a[rand() % (n)]);
34         }
35         for (int i = 0; i != n; ++i) {
36             Q::iterator p = q.push(a[i]);
37             v.push_back(p);
38         }
39         Q::iterator p = q.top();
40
41         comps = 0;
42         for (int i = 0; i != n; ++i) {
43             q.erase(v[i]);
44         }
45         total += comps;
46         q.clear();
47         v.clear();
48     }
49
50     double comp_count = double(total) / double(repetitions);
51     double comp_per_operation = comp_count / double(n);
52     // double factor = comp_per_operation / std::log2(double(n));
53     std::cout << n << " " << comp_per_operation << "\n";
54
55     delete[] a;
56     return 0;
57 }

```

§ 25 *pop-comp.cpp*

```

1 #include <algorithm>
2 #include <cmath>

```

```

3 #include <cstdlib>
4 #include <iostream>
5
6 #ifndef NUMBER
7 #define NUMBER 1000000
8 #endif
9
10 #include "data-structure.i++" // Q comes from here
11
12 int main() {
13     typedef Q::value_type E;
14
15     int const n = NUMBER;
16     E* a = new E[n];
17     srand(1);
18     auto total = comps;
19     Q q;
20
21     int repetitions = 1000000 / n;
22     if (repetitions < 3) {
23         repetitions = 3;
24     }
25
26     for (int k = 0; k < repetitions; ++k) {
27         for (int i = 0; i < n; i++) {
28             a[i] = (E) i;
29         }
30         for (int i = 0; i < n; i++) {
31             std::swap(a[i], a[rand() % (n)]);
32         }
33         for (int i = 0; i != n; ++i) {
34             (void) q.push(a[i]);
35         }
36
37         comps = 0;
38         for (int i = 0; i != n; ++i) {
39             q.pop();
40         }
41         total += comps;
42         q.clear();
43     }
44
45     double comp_count = double(total) / double(repetitions);
46     double comp_per_operation = comp_count / double(n);
47     // double factor = comp_per_operation / std::log2(double(n));
48     std::cout << n << " " << comp_per_operation << "\n";
49
50     delete[] a;
51     return 0;
52 }

```

Makefile

§ 26 *benchmark.mk*

```

1 CXXFLAGS = -DNDEBUG -Wall -std=c++0x -pedantic -x c++ -O3
2 CXX = g++
3
4 implementation-files:= $(wildcard *.i++)
5 data-structures:= $(basename $(implementation-files))
6 time-tests = $(addsuffix .time, $(data-structures))
7 comp-tests = $(addsuffix .comp, $(data-structures))
8 push-time-tests:= $(addsuffix .push, $(time-tests))
9 push-comp-tests:= $(addsuffix .push, $(comp-tests))

```

```

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

```

```
75     ./a.out; \  
76     rm -f ./a.out ; \  
77     done  
78     @rm data-structure.i++  
79  
80 $(increase-comp-tests): %.comp.increase : %.i++  
81     @echo $* "# comp per increase"  
82     @cp $*.i++ data-structure.i++  
83     @for x in $(list) ; do \  
84         $(CXX) $(CXXFLAGS) -DNUMBER=$$x increase-comp.c++;\  
85         ./a.out; \  
86         rm -f ./a.out ; \  
87     done  
88     @rm data-structure.i++  
89  
90 $(erase-comp-tests): %.comp.erase : %.i++  
91     @echo $* "# comp per erase"  
92     @cp $*.i++ data-structure.i++  
93     @for x in $(list) ; do \  
94         $(CXX) $(CXXFLAGS) -DNUMBER=$$x erase-comp.c++;\  
95         ./a.out; \  
96         rm -f ./a.out ; \  
97     done  
98     @rm data-structure.i++  
99  
100 $(pop-comp-tests): %.comp.pop : %.i++  
101     @echo $* "# comp per pop"  
102     @cp $*.i++ data-structure.i++  
103     @for x in $(list) ; do \  
104         $(CXX) $(CXXFLAGS) -DNUMBER=$$x pop-comp.c++;\  
105         ./a.out; \  
106         rm -f ./a.out ; \  
107     done  
108     @rm data-structure.i++
```