

Adaptive heapsort: Source code

Stefan Edelkamp^{1,*}, Amr Elmasry², and Jyrki Katajainen^{2,†}

¹ *TZI, Universität Bremen*

Am Fallturm 1, 28357 Bremen, Germany

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

Abstract. A priority queue is a great tool for solving different kinds of sorting problems. However, a general-purpose priority queue is often an overkill for these applications since the number of elements processed is known beforehand and the operation repertoire to be supported just includes two operations: *insert* and *extract-min*. This report is an electronic appendix to our paper “Two constant-factor-optimal realizations of adaptive heapsort”, where we show how two priority queues—a weak heap and a weak queue—can be specialized and used in adaptive sorting. At the time of writing, the programs developed provide the best performance of all implementations of known adaptive sorting algorithms with respect to the number of element comparisons performed, but due to a poor cache behaviour the programs are not always the fastest when sorting elements for which element moves and element comparisons are cheap. This report together with an accompanying `tar` file gives the source code used in the experiments reported in the paper. Our main conclusion is that the gap between the theory of adaptive sorting and the actual computing practice is still big. By making the source code available, we hope that other researchers can use our code in their experiments and this way confirm that their work actually reduces the gap.

Keywords. Adaptive heapsort, priority queues, weak heaps, weak queues

* Partially supported by DFG grant ED 74/8-1.

† Partially supported by the Danish Natural Science Research Council under contract 09-060411 (project “Generic programming—algorithms and tools”).

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

2012-04-11

Included files

File	Page
§ 1 cartesian-tree.h++	3
§ 2 cartesian-tree-node.h++	4
§ 3 array-based-buffer.h++	5
§ 4 weak-heap-with-buffer.h++	7
§ 5 adaptive-heapsort-with-weak-heap.h++	10
§ 6 adaptive-heapsort-with-weak-heap.i++	11
§ 7 composite-node.h++	12
§ 8 list-based-buffer.h++	14
§ 9 binary-system-tree-inventory.h++	15
§ 10 prefix-array.h++	17
§ 11 weak-queue-with-buffer.h++	19
§ 12 adaptive-heapsort-with-weak-queue.h++	23
§ 13 adaptive-heapsort-with-weak-queue.i++	24
§ 14 introsort.i++	24
§ 15 heapsort.i++	24
§ 16 generator.i++	24
§ 17 running-time.c++	26
§ 18 comparison-count.c++	27
§ 19 code.mk	28

Cartesian tree

```
§ 1 cartesian-tree.h++
```

```

1 /*
2  A Cartesian tree. Due to the lack of parent pointers, on the right
3  spine of the tree, the pointers to the second child are temporarily
4  reverted to point to the parent.
5
6  Authors: Stefan Edelkamp, Jyrki Katajainen © 2010, 2011
7 */
8
9 #ifndef __CPHSTL_CARTESIAN_TREE__
10 #define __CPHSTL_CARTESIAN_TREE__
11
12 #include <cstdlib> // std::size_t
13 #include <vector>
14
15 namespace cphstl {
16   template <typename P, typename C>
17   class cartesian_tree {
18   public:
19
20     typedef P memory_pool_type;
21     typedef C comparator_type;
22     typedef typename P::value_type node_type;
23     typedef typename node_type::element_type element_type;
24     typedef std::size_t size_type;
25
26   protected:
27
28     size_type n;
29     C comparator;
30     node_type* root;
31     node_type* base;
32
33   public:
34
35     node_type* minimum() const {
36       return root;
37     }
38
39     explicit cartesian_tree(P & pool, C const & c = C())
40       : n(pool.size()), comparator(c), root(0), base(&pool[0]) {
41       if (n == 0) {
42         return;
43       }
44       root = base;
45       (*base).first(0);
46       (*base).second(0);
47       for (node_type* u = base + 1; u < base + n; ++u) {
48         node_type* y = u - 1;
49         node_type* z = 0;
50         while (y != 0 && ! comparator((*y).element(), (*u).element())) {
51           node_type* x = (*y).second(); // x is the parent
52           (*y).second(z);
53           z = y;
54           y = x;
55         }
56         if (y == 0) {
57           (*u).first(z);
58           (*u).second(0); // u has no parent
59           root = u;
60         }

```

```

61     else {
62         (*u).first(z);
63         (*u).second(y); // y is the parent
64     }
65 }
66
67 // fix the pointers on the right spine
68
69 node_type* z = 0;
70 node_type* y = base + n - 1;
71 while (y != 0) {
72     node_type* x = (*y).second(); // x is the parent
73     (*y).second(z);
74     z = y;
75     y = x;
76 }
77 root = z;
78 }
79
80 };
81 }
82
83 #endif

```

Weak heap

§ 2 cartesian-tree-node.h++

```

1  /*
2  A Cartesian-tree node
3
4  Author: Jyrki Katajainen © 2011
5  */
6
7  #ifndef __CPHSTL_CARTESIAN_TREE_NODE__
8  #define __CPHSTL_CARTESIAN_TREE_NODE__
9
10 #include <cstdlib> // std::ptrdiff_t
11 #include <list>
12
13 namespace cphstl {
14     template <typename E>
15     class cartesian_tree_node {
16     public:
17
18         typedef E element_type;
19
20         cartesian_tree_node* left;
21         cartesian_tree_node* right;
22         E value;
23
24         explicit cartesian_tree_node() :
25             left(0), right(0) {
26         }
27
28         explicit cartesian_tree_node(E const& v) :
29             left(0), right(0), value(v) {
30         }
31
32         cartesian_tree_node* first() const {
33             return left;
34         }
35
36         cartesian_tree_node* second() const {

```

```

37     return right;
38 }
39
40 E const& element() const {
41     return value;
42 }
43
44 void first(cartesian_tree_node* v) {
45     left = v;
46 }
47
48 void second(cartesian_tree_node* v) {
49     right = v;
50 }
51
52 E& element() {
53     return value;
54 }
55
56 void element(E const& new_value) {
57     value = new_value;
58 }
59
60 };
61 }
62
63 #endif

```

§ 3 *array-based-buffer.h++*

```

1  /*
2  An array implementation of a priority queue
3  - insert: O(1) worst-case time
4  - extract-min: O(n) worst-case time
5  - borrow: O(1) worst-case time
6  - minimum: O(1) worst-case time
7
8  Authors: Stefan Edelkamp, Jyrki Katajainen © 2011
9  */
10
11 #ifndef __CPHSTL_ARRAY_BASED_BUFFER__
12 #define __CPHSTL_ARRAY_BASED_BUFFER__
13
14 #include <algorithm>
15
16 #include <cstdint> // std::size_t
17 #include <vector>
18
19 namespace cphstl {
20     template <typename E, typename C>
21     class array_based_buffer {
22     public:
23
24         // types
25
26         typedef E element_type;
27         typedef C comparator_type;
28         typedef std::vector<E> sequence_type;
29         typedef std::size_t size_type;
30
31     protected:
32
33         // variables
34

```

```

35     C comparator;
36     sequence_type buffer;
37
38 public:
39
40     // structors
41
42     explicit array_based_buffer(C const& c = C(), size_type capacity = 64)
43     : comparator(c), buffer() {
44         buffer.reserve(capacity);
45     }
46
47     ~array_based_buffer() {
48     }
49
50     // accessors
51
52     size_type size() const {
53         return buffer.size();
54     }
55
56     E minimum() const {
57         return buffer[0];
58     }
59
60     // modifiers
61
62     void insert(E const& x) {
63         if (buffer.size() > 0 && comparator(x, buffer[0])) {
64             buffer.push_back(buffer[0]);
65             buffer[0] = x;
66         }
67         else {
68             buffer.push_back(x);
69         }
70     }
71
72     E borrow() {
73         E p = buffer.back();
74         buffer.pop_back();
75         return p;
76     }
77
78     void extract_min() {
79         if (buffer.size() == 1) {
80             buffer.pop_back();
81             return;
82         }
83         E minimum = buffer[1];
84         size_type m = 1;
85         for (size_type i = 2; i < buffer.size(); ++i) {
86             E x = buffer[i];
87             if (comparator(x, minimum)) {
88                 minimum = x;
89                 m = i;
90             }
91         }
92         buffer[0] = buffer[m];
93         buffer[m] = buffer.back();
94         buffer.pop_back();
95     }
96
97 };
98 }
99

```

```
100 #endif
```

§ 4 *weak-heap-with-buffer.h++*

```
1 /*
2  A weak heap for constant-factor-optimal adaptive sorting
3  - insert: O(1) amortized time
4  - extract_min: O(lg n) worst-case time with lg n + O(1) element comparisons
5  - update_min: O(lg n) worst-case time with lg n + O(1) element comparisons
6  - borrow: O(1) worst-case time
7  - minimum: O(1) worst-case time
8
9  Authors: Stefan Edelkamp, Jyrki Katajainen © 2010–2012
10 */
11
12 #ifndef __CPHSTL_WEAK_HEAP_WITH_BUFFER__
13 #define __CPHSTL_WEAK_HEAP_WITH_BUFFER__
14
15 #include <algorithm> // std::max, std::swap
16 #include "array-based-buffer.h++"
17 #include <cmath> // ilogb
18 #include <cstdint> // std::size_t
19 #include <vector>
20 #include <assert.h++>
21
22 extern int ilogb(double) throw();
23
24 namespace cphstl {
25     template <typename E, typename C>
26     class weak_heap {
27     public:
28
29         // types
30
31         typedef E element_type;
32         typedef C comparator_type;
33         typedef std::vector<E> sequence_type;
34         typedef std::vector<unsigned char> bit_sequence_type;
35         typedef cphstl::array_based_buffer<E, C> buffer_type;
36         typedef std::size_t size_type;
37
38     protected:
39
40         // variables
41
42         size_type n;
43         C comparator;
44         sequence_type a;
45         bit_sequence_type r;
46         buffer_type buffer;
47         bool update_heap;
48
49     public:
50
51         // structors
52
53         explicit weak_heap(C const& c = C(), size_type capacity = 10000)
54             : n(0), comparator(c), a(), r(), buffer(c), update_heap(false) {
55             a.reserve(capacity);
56             r.reserve(capacity);
57         }
58
59         ~weak_heap() {
60         }
```

```

61
62 // accessors
63
64 size_type size() const {
65     return n;
66 }
67
68 E minimum() {
69     if (buffer.size() > size_type(ilogb(n)) + 1) {
70         flush_buffer();
71     }
72     if (a.size() == 0) {
73         update_heap = false;
74         return buffer.minimum();
75     }
76     if (buffer.size() == 0) {
77         update_heap = true;
78         return a[0];
79     }
80     E heap_min = a[0];
81     E buffer_min = buffer.minimum();
82     if (comparator(heap_min, buffer_min)) {
83         update_heap = true;
84         return heap_min;
85     }
86     update_heap = false;
87     return buffer_min;
88 }
89
90 // modifiers
91
92 void insert(E const & x) {
93     buffer.insert(x);
94     ++n;
95 }
96
97 E borrow() {
98     --n;
99     if (buffer.size() > 0) {
100         E x = buffer.borrow();
101         return x;
102     }
103     E x = a.back();
104     a.pop_back();
105     r.pop_back();
106     return x;
107 }
108
109 void update_min(E const & x) {
110     if (update_heap) {
111         a[0] = x;
112         sift_down(0, a.size());
113     }
114     else {
115         buffer.extract_min();
116         buffer.insert(x);
117     }
118 }
119
120 void extract_min() {
121     if (update_heap) {
122         if (buffer.size() > 0) {
123             a[0] = buffer.borrow();
124             sift_down(0, a.size());
125         }

```

```

126     else {
127         a[0] = a.back();
128         a.pop_back();
129         r.pop_back();
130         sift_down(0, a.size());
131     }
132 }
133 else {
134     buffer.extract_min();
135 }
136 --n;
137 }
138
139 private:
140
141 size_type distinguished_ancestor(size_type j) const {
142     while (bool(j & 1) == bool(r[j / 2])) {
143         j /= 2;
144     }
145     return j / 2;
146 }
147
148 void sift_up(size_type j) {
149     while (j != 0) {
150         size_type i = distinguished_ancestor(j);
151         if (comparator(a[j], a[i])) {
152             std::swap(a[i], a[j]);
153             r[j] = 1 - r[j];
154         }
155         else {
156             break;
157         }
158         j = i;
159     }
160 }
161
162 void sift_down(size_type j, size_type n) {
163     size_type k = 2 * j + 1 - r[j];
164     if (k >= n)
165         return;
166     while (2 * k + r[k] < n) {
167         k = (2 * k + r[k]);
168     }
169     while (k != j) {
170         if (comparator(a[k], a[j])) {
171             std::swap(a[j], a[k]);
172             r[k] = 1 - r[k];
173         }
174         k /= 2;
175     }
176 }
177
178 void flush_buffer() {
179     size_type right = a.size() + buffer.size() - 1;
180     size_type left = std::max(a.size(), right / 2);
181     while (buffer.size() > 0) {
182         E x = buffer.borrow();
183         a.push_back(x);
184         r.push_back(0);
185     }
186     size_type n = a.size();
187     while (right > 1 + left) {
188         left = left / 2;
189         right = right / 2;
190         for (size_type j = left; j <= right; j++) {

```

```

191     sift_down(j, n);
192     }
193   }
194   for (size_type j = left; j <= right; ++j) {
195     if (j == 0) {
196       continue;
197     }
198     size_type i = distinguished_ancestor(j);
199     sift_down(i, n);
200     sift_up(i);
201   }
202 }
203
204 };
205 }
206
207 #endif

```

Adaptive heapsort using a weak heap

§ 5 *adaptive-heapsort-with-weak-heap.h++*

```

1  /*
2  Adaptive heapsort using a weak heap
3
4  Authors: Stefan Edelkamp, Jyrki Katajainen © 2010, 2011
5  */
6
7  #ifndef __CPHSTL_ADAPTIVE_HEAPSORT_WITH_WEAK_HEAP__
8  #define __CPHSTL_ADAPTIVE_HEAPSORT_WITH_WEAK_HEAP__
9
10 #include <cstdlib> // std::size_t
11 #include "cartesian-tree.h++"
12 #include "cartesian-tree-node.h++"
13 #include <functional> // std::binary_function
14 #include <iterator> // std::iterator_traits
15 #include "weak-heap-with-buffer.h++"
16 #include <vector>
17
18 namespace cphstl {
19   template <typename N, typename C>
20   class indirect_access
21     : public std::binary_function<N*, N*, bool> {
22   protected:
23
24     C less;
25
26   public:
27
28     explicit indirect_access(const C & f)
29       : less(f) {
30     }
31
32     bool operator()(N* const x, N* const y) const {
33       return less((*x).element(), (*y).element());
34     }
35   };
36
37   template <typename R, typename C,
38     typename N = cphstl::cartesian_tree_node<typename std::iterator_traits<R
39     >::value_type> >
39   void adaptive_heapsort(R start, R end, C comparator) {
40     typedef R random_access_iterator;
41     typedef C comparator_type;

```

```

42 typedef N node_type;
43 typedef cphstl::indirect_access<N, C> D;
44 typedef std::size_t size_type;
45 typedef std::vector<N> node_pool_type;
46
47 // create the pool of nodes used by the Cartesian tree
48
49 size_type n = end - start;
50 node_pool_type node_pool(n);
51
52 R i = start;
53 N* p = &node_pool[0];
54 while (i != end) {
55     (*p).element() = *i;
56     ++p;
57     ++i;
58 }
59
60 // construct the Cartesian tree
61
62 cphstl::cartesian_tree<node_pool_type, C> tree(node_pool, comparator);
63
64 // sort the elements
65
66 D indirect_comparator(comparator);
67 cphstl::weak_heap<N*, D> heap(indirect_comparator, n);
68 heap.insert(tree.minimum());
69 R output = start;
70 while (heap.size() > 0) {
71     N* q = heap.minimum();
72     *output = (*q).element();
73     ++output;
74     N* c = (*q).first();
75     N* d = (*q).second();
76     if (c != 0) {
77         heap.update_min(c);
78         if (d != 0) {
79             heap.insert(d);
80         }
81     }
82     else {
83         if (d != 0) {
84             heap.update_min(d);
85         }
86         else {
87             heap.extract_min();
88         }
89     }
90 }
91 }
92 }
93
94 #endif

```

§ 6 adaptive-heapsort-with-weak-heap.i++

```

1 #include "adaptive-heapsort-with-weak-heap.h++"
2 #include "cartesian-tree-node.h++"
3 #include <iterator> // std::iterator_traits
4
5 template <typename R, typename C>
6 void adaptive_heapsort(R start, R end, C comparator) {
7     typedef typename std::iterator_traits<R>::value_type E;
8     typedef cphstl::cartesian_tree_node<E> N;

```

```

9  cphstl::adaptive_heapsort<R, C, N>(start, end, comparator);
10 }
11
12 #define ALGORITHM adaptive_heapsort

```

Weak queue

§ 7 *composite-node.h++*

```

1  /*
2  A node that can be in three data structures
3  - north, south: Cartesian-tree pointers
4  - east, west: weak-heap pointers
5  - east, west: buffer pointers
6
7  Author: Jyrki Katajainen © 2011
8  */
9
10 #ifndef __CPHSTL_COMPOSITE_NODE__
11 #define __CPHSTL_COMPOSITE_NODE__
12
13 namespace cphstl {
14 template <typename E>
15 class composite_node {
16 public:
17
18     typedef E element_type;
19
20     composite_node* north;
21     composite_node* south;
22     composite_node* east;
23     composite_node* west;
24     E value;
25
26     explicit composite_node() :
27         north(0), south(0), east(0), west(0) {
28     }
29
30     explicit composite_node(E const& v) :
31         north(0), south(0), east(0), west(0), value(v) {
32     }
33
34     E const& element() const {
35         return value;
36     }
37
38     composite_node* first() const {
39         return south;
40     }
41
42     composite_node* second() const {
43         return north;
44     }
45
46     composite_node* previous() const {
47         return west;
48     }
49
50     composite_node* next() const {
51         return east;
52     }
53
54     composite_node* left() const {
55         return west;

```

```

56     }
57
58     composite_node* right() const {
59         return east;
60     }
61
62     E& element() {
63         return value;
64     }
65
66     void element(E const& new_value) {
67         value = new_value;
68     }
69
70     void first(composite_node* v) {
71         south = v;
72     }
73
74     void second(composite_node* v) {
75         north = v;
76     }
77
78     void previous(composite_node* v) {
79         west = v;
80     }
81
82     void next(composite_node* v) {
83         east = v;
84     }
85
86     void left(composite_node* v) {
87         west = v;
88     }
89
90     void right(composite_node* v) {
91         east = v;
92     }
93
94     template <typename C>
95     composite_node* link(composite_node* q, C const& comparator) {
96         composite_node* p = this;
97         if (comparator((*q).element(), (*p).element())) {
98             std::swap(q, p);
99         }
100        composite_node* c = (*p).right();
101        (*q).left(c);
102        (*p).left(0);
103        (*p).right(q);
104        return p;
105    }
106
107    void swap_roots(composite_node* q) { // must be outside a tree inventory
108        composite_node* p = this;
109        composite_node* a = (*p).right();
110        composite_node* b = (*q).right();
111        (*q).right(a);
112        (*p).right(b);
113    }
114
115 };
116 }
117
118 #endif

```

§ 8 *list-based-buffer.h++*

```

1  /*
2  A linked-list implementation of a priority queue; the list is
3  singly-linked and circular.
4  – insert: O(1) worst-case time
5  – extract-min: O(n) worst-case time
6  – borrow: O(1) worst-case time
7  – minimum: O(1) worst-case time
8
9  Author: Jyrki Katajainen © 2011
10 */
11
12 #ifndef __CPHSTL_LIST_BASED_BUFFER__
13 #define __CPHSTL_LIST_BASED_BUFFER__
14
15 #include <algorithm>
16
17 #include <cstddef> // std::size_t
18 #include <vector>
19
20 namespace cphstl {
21     template <typename N, typename C>
22     class list_based_buffer {
23     public:
24
25         // types
26
27         typedef N node_type;
28         typedef C comparator_type;
29         typedef std::size_t size_type;
30
31     protected:
32
33         // variables
34
35         size_type n;
36         C comparator;
37         N* head;
38
39     public:
40
41         // structors
42
43         explicit list_based_buffer(C const & c = C())
44             : n(0), comparator(c), head(0) {
45         }
46
47         ~list_based_buffer() {
48         }
49
50         // accessors
51
52         size_type size() const {
53             return n;
54         }
55
56         N* minimum() const {
57             return head;
58         }
59
60         // modifiers
61
62         void insert(N* p) {
63             if (head == 0) {

```

```

64     (*p).next(p);
65     head = p;
66 }
67 else {
68     (*p).next((*head).next());
69     (*head).next(p);
70     if (comparator((*p).element(), (*head).element())) {
71         head = p;
72     }
73 }
74 ++n;
75 }
76
77 N* borrow() {
78     --n;
79     if (n == 0) {
80         N* p = head;
81         (*p).next(0);
82         head = 0;
83         return p;
84     }
85     N* p = (*head).next();
86     (*head).next((*p).next());
87     (*p).next(0);
88     return p;
89 }
90
91 N* extract_min() {
92     --n;
93     if (n == 0) {
94         N* p = head;
95         (*p).next(0);
96         head = 0;
97         return p;
98     }
99     N* p = head;
100    N* q = (*p).next();
101    N* minimum = q;
102    N* r = (*q).next();
103    while (r != head) {
104        if (comparator((*r).element(), (*minimum).element())) {
105            minimum = r;
106        }
107        q = r;
108        r = (*r).next();
109    }
110    (*q).next((*p).next());
111    (*p).next(0);
112    head = minimum;
113    return p;
114 }
115
116 };
117 }
118
119 #endif

```

§ 9 *binary-system-tree-inventory.h++*

```

1 /*
2  Let n the number of elements stored in a weak queue and let
3  <b_{k-1}, ..., b_{-1}, b_{-0}> be the binary representation of n, where
4  b_{k-1} is the most significant digit. This inventory stores a
5  perfect weak heap of rank j if and only if b_{-j} = 1, as proposed by

```

```

6  Vuillemin in his seminal paper from 1978.
7
8  Author: Jyrki Katajainen © 2010, 2011
9  */
10
11 #ifndef __CPHSTL_BINARY_SYSTEM_TREE_INVENTORY__
12 #define __CPHSTL_BINARY_SYSTEM_TREE_INVENTORY__
13
14 #include <climits> // CHAR_BIT
15 #include <cmath> // ilogb
16 #include <cstdlib> // std::size_t
17
18 extern int ilogb(double) throw();
19
20 namespace cphstl {
21     template<typename N, typename C>
22     class binary_system_tree_inventory {
23     public:
24
25         typedef N node_type;
26         typedef C comparator_type;
27         typedef std::size_t size_type;
28
29     protected:
30
31         static const size_type address_size = CHAR_BIT * sizeof(N*);
32         size_type n;
33         C comparator;
34         N* header[address_size];
35
36     public:
37
38         binary_system_tree_inventory(C const& c = C())
39             : n(0), comparator(c) {
40             for (size_type i = 0; i != __end__(); ++i) {
41                 header[i] = 0;
42             }
43         }
44
45         ~binary_system_tree_inventory() {
46         }
47
48         size_type __rank__(N* const p) const {
49             size_type r = 0;
50             while (r != __end__()) {
51                 if (header[r] == p) {
52                     return r;
53                 }
54                 ++r;
55             }
56             return __end__();
57         }
58
59         size_type __first__() const {
60             if (n == 0) {
61                 return __end__();
62             }
63             size_type r = 0;
64             size_type mask = n;
65             while (r != __end__()) {
66                 if ((mask & 1) == 1) {
67                     return r;
68                 }
69                 ++r;
70                 mask >>= 1;

```

```

71     }
72     return __end__();
73 }
74
75 size_type __last__() const {
76     if (n == 0) {
77         return __end__();
78     }
79     return size_type(ilogb(n));
80 }
81
82 size_type __end__() const {
83     return address_size;
84 }
85
86 N* __access__(size_type rank) const {
87     return header[rank];
88 }
89
90 size_type __next__(size_type current) const {
91     size_type r = current + 1;
92     size_type mask = n >> r;
93     if (mask == 0) {
94         return __end__();
95     }
96     while (r != __end__()) {
97         if ((mask & 1) == 1) {
98             return r;
99         }
100        ++r;
101        mask >>= 1;
102    }
103    return __end__();
104 }
105
106 void insert(size_type rank, N* p) {
107     N* q = header[rank];
108     n += (1 << rank);
109     while (q != 0) {
110         p = (*p).link(q, comparator);
111         header[rank] = 0;
112         ++rank;
113         q = header[rank];
114     }
115     header[rank] = p;
116 }
117
118 void extract(size_type rank, N* q) {
119     n -= (1 << rank);
120     header[rank] = 0;
121 }
122
123 void update(size_type rank, N* q) {
124     header[rank] = q;
125 }
126
127 };
128 }
129
130 #endif

```

§ 10 *prefix-array.h++*

1 /*

```

2  An array storing the prefix-minimum pointers for a weak queue
3
4  Author: Jyrki Katajainen © 2011
5  */
6
7  #ifndef __CPHSTL_PREFIX_ARRAY__
8  #define __CPHSTL_PREFIX_ARRAY__
9
10 #include <climits> // CHAR_BIT
11 #include <cmath> // ilogb
12 #include <cstddef> // std::size_t
13
14 extern int ilogb(double) throw();
15
16 namespace cphstl {
17     template <typename N, typename C>
18     class prefix_array {
19     public:
20
21         typedef N node_type;
22         typedef C comparator_type;
23         typedef typename node_type::element_type element_type;
24         typedef std::size_t size_type;
25
26     protected:
27
28         static const size_type address_size = CHAR_BIT * sizeof(N*);
29         size_type n;
30         C comparator;
31         N* prefix_minima[address_size];
32
33     public:
34
35         explicit prefix_array(C const & c = C())
36             : n(0), comparator(c), prefix_minima() {
37             for (size_type i = 0; i < address_size; ++i) {
38                 prefix_minima[i] = 0;
39             }
40         }
41
42         template <typename T>
43         bool self_loop(size_type rank, T const & tree_inventory) {
44             N* p = tree_inventory.__access__(rank);
45             return prefix_minima[rank] == p;
46         }
47
48         N* minimum() const {
49             if (n == 0) {
50                 return 0;
51             }
52             size_type most_significant_one = ilogb(n);
53             return prefix_minima[most_significant_one];
54         }
55
56         void update(size_type rank, N* to_point_here) {
57             N* old_value = prefix_minima[rank];
58             prefix_minima[rank] = to_point_here;
59             if (old_value == 0 && to_point_here != 0) {
60                 n += (1 << rank);
61             }
62             else if (old_value != 0 && to_point_here == 0) {
63                 n -= (1 << rank);
64             }
65         }
66     };

```

```

67     template <typename T>
68     void refresh(size_type forward_from, T const& tree_inventory) {
69         size_type old_max_rank = 0;
70         if (n != 0) {
71             old_max_rank = ilogb(n);
72         }
73         for (size_type i = forward_from; i <= old_max_rank; ++i) {
74             prefix_minima[i] = 0;
75         }
76         n &= (1 << forward_from) - 1; // unset high bits
77         N* minimum = 0;
78         for (size_type i = 0; i < forward_from; ++i) {
79             if (tree_inventory.__access__(i) != 0) {
80                 minimum = prefix_minima[i];
81             }
82         }
83         size_type new_max_rank = tree_inventory.__last__();
84         if (new_max_rank == tree_inventory.__end__()) {
85             return;
86         };
87         for (size_type i = forward_from; i <= new_max_rank; ++i) {
88             N* p = tree_inventory.__access__(i);
89             if (p == 0) {
90                 continue;
91             }
92             n += (1 << i); // set bit
93             if (minimum == 0) {
94                 prefix_minima[i] = p;
95                 minimum = p;
96             }
97             else if (comparator((*p).element(), (*minimum).element())) {
98                 prefix_minima[i] = p;
99                 minimum = p;
100             }
101             else {
102                 prefix_minima[i] = minimum;
103             }
104         }
105     }
106 };
107 }
108 }
109
110 #endif

```

§ 11 weak-queue-with-buffer.h++

```

1  /*
2  A weak queue for constant-factor-optimal adaptive sorting
3  – insert:  $O(1)$  amortized time
4  – extract_min:  $O(\lg n)$  worst-case time with  $\lg n + O(1)$  element comparisons
5  – update_min:  $O(\lg n)$  worst-case time with  $\lg n + O(1)$  element comparisons
6  – borrow:  $O(\lg n)$  worst-case time with no element comparisons
7  – minimum:  $O(1)$  worst-case time
8
9  Operations to be supported by the tree inventory: constructor,
10 __rank__, __first__, __access__, __next__, __last__, __end__,
11 insert, extract, update.
12
13 Author: Jyrki Katajainen © 2010, 2011
14 */
15
16 #ifndef __CPHSTL_WEAK_QUEUE_WITH_BUFFER__
17 #define __CPHSTL_WEAK_QUEUE_WITH_BUFFER__

```

```

18
19 #include <algorithm>
20
21 #include "binary-system-tree-inventory.h++"
22 #include <cmath> // ilogb
23 #include <cstdlib> // std::size_t
24 #include "list-based-buffer.h++"
25 #include "prefix-array.h++"
26 #include <vector>
27
28 extern int ilogb(double) throw();
29
30 namespace cphstl {
31     template <typename N, typename C>
32     class weak_queue {
33     public:
34
35         // types
36
37         typedef N node_type;
38         typedef C comparator_type;
39         typedef typename N::element_type element_type;
40         typedef cphstl::list_based_buffer<N, C> buffer_type;
41         typedef cphstl::prefix_array<N, C> prefix_array_type;
42         typedef cphstl::binary_system_tree_inventory<N, C> tree_inventory_type;
43         typedef std::size_t size_type;
44
45     protected:
46
47         // variables
48
49         size_type n;
50         comparator_type comparator;
51         buffer_type buffer;
52         tree_inventory_type tree_inventory;
53         prefix_array_type prefix_array;
54
55     public:
56
57         // structors
58
59         explicit weak_queue(C const & c = C())
60             : n(0), comparator(c), buffer(c), tree_inventory(c), prefix_array(c) {
61         }
62
63         ~weak_queue() {
64         }
65
66         // accessors
67
68         size_type size() const {
69             return n;
70         }
71
72         N* minimum() const {
73             N* p = buffer.minimum();
74             N* q = prefix_array.minimum();
75             if (p == 0) {
76                 return q;
77             }
78             if (q == 0) {
79                 return p;
80             }
81             if (comparator((*q).element(), (*p).element())) {
82                 return q;

```

```

83     }
84     return p;
85 }
86
87 // modifiers
88
89 void insert(N* p) {
90     buffer.insert(p);
91     ++n;
92     if (buffer.size() > size_type(ilogb(n)) + 1) {
93         flush_buffer();
94     }
95 }
96
97 N* borrow() {
98     --n;
99     if (buffer.size() > 0) {
100         N* p = buffer.borrow();
101         return p;
102     }
103     size_type small = tree_inventory.__first__();
104     N* p = tree_inventory.__access__(small);
105     if (n == 0) {
106         tree_inventory.extract(0, p);
107         prefix_array.update(0, 0);
108         return p;
109     }
110     if (small == 0) {
111         size_type large = tree_inventory.__next__(small);
112         N* next = tree_inventory.__access__(large);
113         tree_inventory.extract(0, p);
114         prefix_array.update(0, 0);
115         if (!prefix_array.self_loop(large, tree_inventory)) {
116             tree_inventory.extract(large, next);
117             (*p).swap_roots(next);
118             tree_inventory.insert(large, p);
119             return next;
120         }
121         return p;
122     }
123     tree_inventory.extract(small, p);
124     prefix_array.update(small, 0);
125     N* q = (*p).right();
126     (*p).right(0);
127     --small;
128     while (small > 0) {
129         N* r = (*q).left();
130         (*q).left(0);
131         tree_inventory.insert(small, q);
132         prefix_array.update(small, p);
133         q = r;
134         --small;
135     }
136     tree_inventory.insert(0, p);
137     prefix_array.update(0, p);
138     return q;
139 }
140
141 void update_min(N* r) {
142     N* p = buffer.minimum();
143     N* q = prefix_array.minimum();
144     if (q == 0 || (p != 0 && ! comparator((*q).element(), (*p).element()))) {
145         (void) buffer.extract_min();
146         buffer.insert(r);
147         return;

```

```

148     }
149     size_type rank = tree_inventory.__rank__(q);
150     N* t = link_forest(r, q);
151     tree_inventory.update(rank, t);
152     prefix_array.refresh(rank, tree_inventory);
153 }
154
155 N* extract_min() {
156     N* p = buffer.minimum();
157     N* q = prefix_array.minimum();
158     if (q == 0 || (p != 0 && ! comparator((*q).element(), (*p).element()))) {
159         p = buffer.extract_min();
160         --n;
161         return p;
162     }
163     N* replacement;
164     if (buffer.size() > 0) {
165         --n;
166         replacement = buffer.borrow();
167     }
168     else {
169         replacement = borrow();
170     }
171     if (n == 0) {
172         return replacement;
173     }
174     size_type rank = tree_inventory.__rank__(q);
175     N* t = link_forest(replacement, q);
176     tree_inventory.update(rank, t);
177     prefix_array.refresh(rank, tree_inventory);
178     return q;
179 }
180
181 protected:
182
183 void flush_buffer() {
184     while (buffer.size() > 0) {
185         N* p = buffer.borrow();
186         tree_inventory.insert(0, p);
187     }
188     prefix_array.refresh(0, tree_inventory);
189 }
190
191 N* link_forest(N* singleton, N* root) {
192     N* r = root;
193     N* s = (*root).right();
194     while (s != 0) {
195         N* t = (*s).left();
196         (*s).left(r);
197         r = s;
198         s = t;
199     }
200     N* t = singleton;
201     s = r;
202     while (s != root) {
203         r = (*s).left();
204         t = (*t).link(s, comparator);
205         s = r;
206     }
207     return t;
208 }
209
210 };
211 }
212

```

```
213 #endif
```

Adaptive heapsort using a weak queue

§ 12 *adaptive-heapsort-with-weak-queue.hpp*

```
1 /*
2  Adaptive heapsort using a weak queue
3
4  Authors: Stefan Edelkamp, Jyrki Katajainen © 2010, 2011
5 */
6
7 #ifndef __CPHSTL_ADAPTIVE_HEAPSORT_WITH_WEAK_QUEUE__
8 #define __CPHSTL_ADAPTIVE_HEAPSORT_WITH_WEAK_QUEUE__
9
10 #include <stddef> // std::size_t
11 #include <cstdlib>
12 #include "cartesian-tree.hpp"
13 #include <cmath> // srand, rand
14 #include "composite-node.hpp"
15 #include <functional>
16 #include <iterator> // std::iterator_traits
17 #include "weak-queue-with-buffer.hpp"
18 #include <vector>
19
20 namespace cphstl {
21     template <typename R, typename C,
22             typename N = cphstl::composite_node<typename std::iterator_traits<R>::
23                 value_type>>
24     void adaptive_heapsort(R start, R end, C comparator) {
25         typedef R random_access_iterator;
26         typedef C comparator_type;
27         typedef N node_type;
28         typedef std::size_t size_type;
29         typedef std::vector<N> node_pool_type;
30
31         // create the pool of nodes used by all the data structures
32
33         size_type n = end - start;
34         node_pool_type node_pool(n);
35
36         R i = start;
37         N* p = &node_pool[0];
38         while (i != end) {
39             (*p).element() = *i;
40             ++p;
41             ++i;
42         }
43
44         // construct the Cartesian tree
45
46         cphstl::cartesian_tree<node_pool_type, C> tree(node_pool, comparator);
47
48         // sort the elements
49
50         cphstl::weak_queue<N, C> queue(comparator);
51         queue.insert(tree.minimum());
52         R output = start;
53         while (queue.size() > 0) {
54             N* q = queue.minimum();
55             *output = (*q).element();
56             ++output;
57             N* c = (*q).first();
58             N* d = (*q).second();
```

```

58     if (c != 0) {
59         (void) queue.update_min(c);
60         if (d != 0) {
61             queue.insert(d);
62         }
63     }
64     else {
65         if (d != 0) {
66             (void) queue.update_min(d);
67         }
68         else {
69             (void) queue.extract_min();
70         }
71     }
72 }
73 }
74 }
75
76 #endif

```

§ 13 *adaptive-heapsort-with-weak-queue.i++*

```

1 #include "adaptive-heapsort-with-weak-queue.h++"
2 #include "composite-node.h++"
3 #include <iterator> // std::iterator_traits
4
5 template <typename R, typename C>
6 void adaptive_heapsort(R start, R end, C comparator) {
7     typedef typename std::iterator_traits<R>::value_type E;
8     typedef cphstl::composite_node<E> N;
9     cphstl::adaptive_heapsort<R, C, N>(start, end, comparator);
10 }
11
12 #define ALGORITHM adaptive_heapsort

```

Introsort

§ 14 *introsort.i++*

```

1 #include <algorithm>
2
3 #define ALGORITHM std::sort

```

Heapsort

§ 15 *heapsort.i++*

```

1 #include <algorithm>
2
3 template <typename R, typename C>
4 void heapsort(R start, R end, C comparator) {
5     std::partial_sort(start, end, end, comparator);
6 }
7
8 #define ALGORITHM heapsort

```

Benchmark drivers

§ 16 *generator.i++*

```

1 #include <algorithm> // std::random_shuffle
2
3 template <typename V>
4 void random_sequence(V& data) {
5     typedef std::size_t size_type;
6     typedef typename V::value_type E;
7
8     srand(10);
9     size_type n = data.size();
10    for (size_type i = 0; i < n; ++i) {
11        data[i] = (E) rand();
12    }
13 }
14
15 template <typename V>
16 void type_one(V& data, long long inv = 100000) {
17     typedef std::size_t size_type;
18     typedef typename V::value_type E;
19
20     size_type n = data.size();
21     for (size_type i = 0; i < n; ++i) {
22         data[i] = E(i);
23     }
24
25     for (long long i=0;i<inv;i++) {
26         size_type r = rand() % (n-1);
27         std::swap(data[r],data[r+1]);
28     }
29 }
30
31 template <typename V>
32 void type_two(V& data, long long k = 100) {
33     typedef std::size_t size_type;
34     typedef typename V::value_type E;
35
36     size_type n = data.size();
37     for (size_type i = 0; i < n; ++i) {
38         data[i] = E(i);
39     }
40
41     k = (n/k);
42     for (size_type i=0;i<=k;i++) {
43         size_type offset = 1;
44         for (size_type j=i*n/k + 1; (j<n) && (j<(i+1)*n/k);j++) {
45             size_type r = rand() % offset;
46             std::swap(data[j],data[j - r-1]);
47             offset++;
48         }
49     }
50     size_type* indices = new size_type[n/k];
51     size_type r = 0;
52     for (size_type i=0;i<(n/k);i++)
53         indices[r++] = i * k + (rand() % k);
54     for (size_type i=1;i<(n/k);i++)
55         std::swap(data[indices[i]],data[indices[rand()%i]]);
56     delete indices;
57 }
58
59 template <typename V>
60 void random_permutation(V& data) {
61     typedef std::size_t size_type;
62     typedef typename V::value_type E;
63
64     size_type n = data.size();
65     for (size_type i = 0; i < n; ++i) {

```

```

66     data[i] = E(i);
67 }
68 std::random_shuffle(data.begin(), data.end());
69 }
70
71 template <typename V>
72 void zero_sequence(V& data) {
73     typedef std::size_t size_type;
74     typedef typename V::value_type E;
75
76     size_type n = data.size();
77     for (size_type i = 0; i < n; ++i) {
78         data[i] = E(0);
79     }
80 }

```

§ 17 *running-time.cpp*

```

1 #include <algorithm> // std::copy, std::equal, std::sort
2 #include "algorithm.i++" // ALGORITHM is defined here
3 #include <cassert>
4 #include <cstdlib> // std::size_t
5 #include <functional>
6 #include "generator.i++" // random_permutation
7 #include <sys/param.h>
8 #include <sys/times.h>
9 #include <sys/types.h>
10 #include <vector>
11
12 #ifndef NUMBER
13 #define NUMBER 10
14 #endif
15
16 #ifndef GENERATOR
17 #define GENERATOR random_permutation
18 #endif
19
20 int main(int argc, char *argv[]) {
21     typedef int E;
22     typedef std::less<E> C;
23     typedef std::vector<E> D;
24     typedef std::size_t size_type;
25
26     size_type n = NUMBER;
27     std::cout << n << " ";
28
29     D data(n);
30     GENERATOR(data);
31
32     D copy(n);
33     std::copy(data.begin(), data.end(), copy.begin());
34
35     struct tms from, to;
36     times(&from);
37
38     ALGORITHM(data.begin(), data.end(), C());
39
40     times(&to);
41     std::cout << ((float)(to.tms_utime - from.tms_utime)) / (HZ);
42     std::cout << std::endl;
43
44     std::sort(copy.begin(), copy.end(), C());
45
46     return 0;

```

47 }

§ 18 *comparison-count.cpp*

```

1 #include <algorithm> // std::copy, std::equal, std::sort
2 #include "algorithm.i++" // ALGORITHM is defined here
3 #include <cassert>
4 #include <cstdint> // std::size_t
5 #include <functional> // std::binary_function
6 #include "generator.i++" // random_permutation
7 #include <sys/param.h>
8 #include <sys/times.h>
9 #include <sys/types.h>
10 #include <vector>
11
12 long long comps = 0;
13
14 template <typename T>
15 class counting_comparator
16 : public std::binary_function<T, T, bool> {
17 public:
18
19     bool operator()(T const& a, T const& b) const {
20         ++comps;
21         return a < b;
22     }
23 };
24
25 #ifndef NUMBER
26 #define NUMBER 10
27 #endif
28
29 #ifndef GENERATOR
30 #define GENERATOR random_permutation
31 #endif
32
33 int main(int argc, char *argv[]) {
34     typedef int E;
35     typedef counting_comparator<E> C;
36     typedef std::vector<E> D;
37     typedef std::size_t size_type;
38
39     size_type n = NUMBER;
40     std::cout << n << " ";
41
42     D data(n);
43     GENERATOR(data);
44
45     D copy(n);
46     std::copy(data.begin(), data.end(), copy.begin());
47
48     comps = 0;
49
50     ALGORITHM(data.begin(), data.end(), C());
51
52     std::cout << (float) comps / n;
53     std::cout << std::endl;
54
55     std::sort(copy.begin(), copy.end(), C());
56
57     return 0;
58 }

```

Makefile

§ 19 *code.mk*

```

1 CXXFLAGS = -Wall -pedantic -ansi -Wno-long-long -x c++ -std=c++0x -g -DDEBUG
2 REALTEST = -Wall -pedantic -ansi -Wno-long-long -x c++ -std=c++0x -DNDEBUG -O3
3 IFLAGS = -I. -I $(CPHSTL)/Source/Assert/Code
4 CXX = g++
5
6 unittests: cartesian-tree-test list-based-buffer-test binary-system-tree-inventory-
    test weak-queue-test adaptive-heapsort-with-weak-queue-test array-based-buffer-
    test weak_heap_test adaptive-heapsort-with-weak-heap-test
7     @echo "Unittests passed!"
8
9 implementations:= $(wildcard *.i++)
10 algorithms:= $(basename $(implementations))
11 time-tests = $(addsuffix .time, $(algorithms))
12 comp-tests = $(addsuffix .comp, $(algorithms))
13
14 test = 500000 5000000 # 50000000
15 data-type = "random_permutation" # "zero_sequence" "type_one" "type_two"
16
17 $(time-tests): %.time : %.i++
18     @echo $* "running time (s)"
19     @cp $*.i++ algorithm.i++
20     @for d in $(data-type) ; do \
21         echo $$d; \
22         for x in $(test) ; do \
23             $(CXX) $(REALTEST) -DNUMBER=$$x -DGENERATOR=$$d $(IFLAGS) running-time.c
24             ++; \
25             ./a.out; \
26             rm -f ./a.out ; \
27         done \
28     @rm algorithm.i++
29
30 $(comp-tests): %.comp : %.i++
31     @echo $* "# comparisons per element"
32     @cp $*.i++ algorithm.i++
33     @for d in $(data-type) ; do \
34         echo $$d; \
35         for x in $(test) ; do \
36             $(CXX) $(REALTEST) -DNUMBER=$$x -DGENERATOR=$$d $(IFLAGS) comparison-
37             count.c++; \
38             ./a.out; \
39             rm -f ./a.out ; \
40         done \
41     @rm algorithm.i++
42
43 cartesian-tree-test:
44     $(CXX) $(CXXFLAGS) $(IFLAGS) -DUNITTEST_CARTESIAN_TREE -DDEBUG cartesian-
45     tree.h++
46     ./a.out > /dev/null
47
48 list-based-buffer-test:
49     $(CXX) $(CXXFLAGS) $(IFLAGS) -DUNITTEST_LIST_BASED_BUFFER -DDEBUG list-based-
50     -buffer.h++
51     ./a.out > /dev/null
52
53 array-based-buffer-test:
54     $(CXX) $(CXXFLAGS) $(IFLAGS) -DUNITTEST_ARRAY_BASED_BUFFER -DDEBUG array-
55     based-buffer.h++
56     ./a.out > /dev/null

```

```

54
55 binary-system-tree-inventory-test:
56     $(CXX) $(CXXFLAGS) $(IFLAGS) -DUNITTEST_BINARY_SYSTEM_TREE_INVENTORY -DDEBUG
        binary-system-tree-inventory.h++
57     ./a.out > /dev/null
58     @rm -f a.out
59
60 list = 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160
61
62 weak-queue-test:
63     @for x in $(list) ; do \
64         $(CXX) $(CXXFLAGS) -DNUMBER=$$x $(IFLAGS) -
            DUNITTEST_WEAK_QUEUE_WITH_BUFFER -DDEBUG weak-queue-with-buffer.h++;
        \
65         ./a.out; \
66         rm -f ./a.out ; \
67     done
68
69 adaptive-heapsort-with-weak-queue-test:
70     @for x in $(list) ; do \
71         $(CXX) $(CXXFLAGS) -DNUMBER=$$x $(IFLAGS) -
            DUNITTEST_ADAPTIVE_HEAPSORT_WITH_WEAK_QUEUE -DDEBUG adaptive-heapsort
            -with-weak-queue.h++; \
72         ./a.out; \
73     done
74
75 weak-heap-test:
76     @for x in $(list) ; do \
77         $(CXX) $(CXXFLAGS) -DNUMBER=$$x $(IFLAGS) -DUNITTEST_WEAK_HEAP_WITH_BUFFER
            -DDEBUG weak-heap-with-buffer.h++; \
78         ./a.out; \
79         rm -f ./a.out ; \
80     done
81
82 adaptive-heapsort-with-weak-heap-test:
83     @for x in $(list) ; do \
84         $(CXX) $(CXXFLAGS) -DNUMBER=$$x $(IFLAGS) -
            DUNITTEST_ADAPTIVE_HEAPSORT_WITH_WEAK_HEAP -DDEBUG adaptive-heapsort-
            with-weak-heap.h++; \
85         ./a.out; \
86     done
87
88 script-directory=$(CPHSTL)/Script/Build-system
89 input-directory=$(CPHSTL)/Source
90 output-directory=Report-2011-1
91
92 package:
93     export PYTHONPATH=".$(script-directory)"; python $(script-directory)/build
        .py --source=$(input-directory)/Adaptive-heapsort,$(input-directory)/
        Assert/Code --target=$(output-directory) --clean --retain=NDEBUG --
        exclude=$(output-directory),$(output-directory)~,Backup,Experimental,
        Stefan-* package.txt
94     cp doc.tex $(output-directory)/
95     cp report.mk $(output-directory)/
96     cd $(output-directory); ln -s report.mk makefile
97     @mv $(output-directory)~/doc.www $(output-directory)/ 2>/dev/null
98     @mv $(output-directory)~/cvsignore $(output-directory)/ 2>/dev/null
99     @mv $(output-directory)~/CVS $(output-directory)/ 2>/dev/null
100
101 clean:
102     @rm -vf *~ a.out core

```