Adaptive heapsort: Source code

Stefan Edelkamp[1,*], Amr Elmasry[2], and Jyrki Katajainen[3,†]

[1] *TZI, Universität Bremen*
Am Fallturm 1, 28357 Bremen, Germany
[2] *Department of Computer Engineering and Systems, Alexandria University, Alexandria 21544, Egypt*
[3] *Department of Computer Science, University of Copenhagen*
Universitetsparken 5, 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").

CPH STL Report 2011-1, June 2011. Revised April 2012, November 2012, June 2015.

Copyright notice

Release date

2015-06-11

Included files

Acknowledgement

We thank Piotr Cerobski for reporting an error in our build system.

Cartesian tree

§ 1 *cartesian-tree.h++*

```
/*
  A Cartesian tree. Due to the lack of parent pointers, on the right
  spine of the tree, the pointers to the second child are temporarily
  reverted to point to the parent.

  Authors: Stefan Edelkamp, Jyrki Katajainen © 2010, 2011
*/

#include <cstddef> // std::size_t
#include <vector>

namespace cphstl {

  template <typename P, typename C>
  class cartesian_tree {
  public:

    typedef P memory_pool_type;
    typedef C comparator_type;
    typedef typename P::value_type node_type;
    typedef typename node_type::element_type element_type;
    typedef std::size_t size_type;

  protected:

    size_type n;
    C comparator;
    node_type* root;
    node_type* base;

  public:

    node_type* minimum() const {
      return root;
    }

    explicit cartesian_tree(P & pool, C const & c = C())
      : n(pool.size()), comparator(c), root(0), base( &pool[0]) {
      if (n == 0) {
        return;
      }
      root = base;
      (*base).first(0);
      (*base).second(0);
      for (node_type* u = base + 1; u < base + n; ++u) {
        node_type* y = u - 1;
        node_type* z = 0;
        while (y != 0 && ! comparator((*y).element(), (*u).element())) {
          node_type* x = (*y).second(); // x is the parent
          (*y).second(z);
          z = y;
          y = x;
        }
        if (y == 0) {
          (*u).first(z);
          (*u).second(0); // u has no parent
          root = u;
        }
        else {
          (*u).first(z);
```

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

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 <cstddef> // std::ptrdiff_t
11 #include <list>
12
13 namespace cphstl {
14
15   template <typename E>
16   class cartesian_tree_node {
17   public:
18
19     typedef E element_type;
20
21     cartesian_tree_node* left;
22     cartesian_tree_node* right;
23     E value;
24
25     explicit cartesian_tree_node() :
26       left(0), right(0) {
27     }
28
29     explicit cartesian_tree_node(E const & v) :
30       left(0), right(0), value(v) {
31     }
32
33     cartesian_tree_node* first() const {
34       return left;
35     }
36
37     cartesian_tree_node* second() const {
38       return right;
39     }
40
41     E const & element() const {
```

```
42      return value;
43    }
44
45    void first(cartesian_tree_node* v) {
46      left = v;
47    }
48
49    void second(cartesian_tree_node* v) {
50      right = v;
51    }
52
53    E& element() {
54      return value;
55    }
56
57    void element(E const& new_value) {
58      value = new_value;
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 #include <algorithm>
12 #include <cstddef> // std::size_t
13 #include <vector>
14
15 namespace cphstl {
16
17   template <typename E, typename C>
18   class array_based_buffer {
19   public:
20
21     // types
22
23     typedef E element_type;
24     typedef C comparator_type;
25     typedef std::vector<E> sequence_type;
26     typedef std::size_t size_type;
27
28   protected:
29
30     // variables
31
32     C comparator;
33     sequence_type buffer;
34
35   public:
36
37     // structors
38
39     explicit array_based_buffer(C const& c = C(), size_type capacity = 64)
```

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

§ 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 #include <algorithm> // std::max, std::swap
13 #include "array−based−buffer.h++"
14 #include <cmath> // ilogb
15 #include <cstddef> // std::size_t
16 #include <vector>
17
18 extern int ilogb(double) throw();
19
20 namespace cphstl {
21
22   template <typename E, typename C>
23   class weak_heap {
24   public:
25
26     // types
27
28     typedef E element_type;
29     typedef C comparator_type;
30     typedef std::vector<E> sequence_type;
31     typedef std::vector<unsigned char> bit_sequence_type;
32     typedef cphstl::array_based_buffer<E, C> buffer_type;
33     typedef std::size_t size_type;
34
35   protected:
36
37     // variables
38
39     size_type n;
40     C comparator;
41     sequence_type a;
42     bit_sequence_type r;
43     buffer_type buffer;
44     bool minimum_in_heap;
45
46   public:
47
48     // structors
49
50     explicit weak_heap(C const& c = C(), size_type capacity = 10000)
51       : n(0), comparator(c), a(), r(), buffer(c), minimum_in_heap(false) {
52       a.reserve(capacity);
53       r.reserve(capacity);
54     }
55
56     ~weak_heap() {
57     }
58
59     // accessors
60
61     size_type size() const {
62       return n;
63     }
64
65     E minimum() {
66       if (buffer.size() > size_type(ilogb(n)) + 1) {
67         flush_buffer();
68         minimum_in_heap = true;
69       }
70       if (a.size() == 0) {
71         return buffer.minimum();
```

```
72        }
73        if (buffer.size() == 0 || minimum_in_heap) {
74          return a[0];
75        }
76        E heap_min = a[0];
77        E buffer_min = buffer.minimum();
78        if (comparator(heap_min, buffer_min)) {
79          minimum_in_heap = true;
80          return heap_min;
81        }
82        return buffer_min;
83      }
84
85      // modifiers
86
87      void insert(E const & x) {
88        buffer.insert(x);
89        minimum_in_heap = false;
90        ++n;
91      }
92
93      E borrow() {
94        --n;
95        if (buffer.size() > 0) {
96          E x = buffer.borrow();
97          return x;
98        }
99        E x = a.back();
100       a.pop_back();
101       r.pop_back();
102       minimum_in_heap = true;
103       return x;
104     }
105
106     void update_min(E const & x) {
107       if (buffer.size() == 0) {
108         a[0] = x;
109         sift_down(0, a.size());
110         minimum_in_heap = true;
111         return;
112       }
113       if (a.size() == 0) {
114         buffer.extract_min();
115         buffer.insert(x);
116         minimum_in_heap = false;
117         return;
118       }
119       if (minimum_in_heap) {
120         a[0] = x;
121         sift_down(0, a.size());
122         minimum_in_heap = false;
123         return;
124       }
125       E heap_min = a[0];
126       E buffer_min = buffer.minimum();
127       if (comparator(heap_min, buffer_min)) {
128         a[0] = x;
129         sift_down(0, a.size());
130         minimum_in_heap = false;
131         return;
132       }
133       buffer.extract_min();
134       buffer.insert(x);
135       minimum_in_heap = false;
136     }
```

```
137
138     void extract_min() {
139       --n;
140       if (buffer.size() == 0) {
141         a[0] = a.back();
142         a.pop_back();
143         r.pop_back();
144         sift_down(0, a.size());
145         minimum_in_heap = true;
146         return;
147       }
148       if (a.size() == 0) {
149         buffer.extract_min();
150         return;
151       }
152       if (minimum_in_heap) {
153         a[0] = buffer.borrow();
154         sift_down(0, a.size());
155         minimum_in_heap = false;
156         return;
157       }
158       E heap_min = a[0];
159       E buffer_min = buffer.minimum();
160       if (comparator(heap_min, buffer_min)) {
161         a[0] = buffer.borrow();
162         sift_down(0, a.size());
163         minimum_in_heap = false;
164         return;
165       }
166       buffer.extract_min();
167     }
168
169   private:
170
171     size_type distinguished_ancestor(size_type j) const {
172       while (bool(j & 1) == bool(r[j / 2])) {
173         j /= 2;
174       }
175       return j / 2;
176     }
177
178     void sift_up(size_type j) {
179       while (j != 0) {
180         size_type i = distinguished_ancestor(j);
181         if (comparator(a[j], a[i])) {
182           std::swap(a[i], a[j]);
183           r[j] = 1 - r[j];
184         }
185         else {
186           break;
187         }
188         j = i;
189       }
190     }
191
192     void sift_down(size_type j, size_type n) {
193       size_type k = 2 * j + 1 - r[j];
194       if (k >= n) {
195         return;
196       }
197       while (2 * k + r[k] < n) {
198         k = (2 * k + r[k]);
199       }
200       while (k != j) {
201         if (comparator(a[k], a[j])) {
```

```
202           std::swap(a[j], a[k]);
203           r[k] = 1 - r[k];
204         }
205         k /= 2;
206       }
207     }
208
209     void flush_buffer() {
210       size_type right = a.size() + buffer.size() - 1;
211       size_type left = std::max(a.size(), right / 2 + 1);
212       while (buffer.size() > 0) {
213         E x = buffer.borrow();
214         a.push_back(x);
215         r.push_back(0);
216       }
217       size_type n = a.size();
218       while (right > 1 + left) {
219         left = left / 2;
220         right = right / 2;
221         for (size_type j = left; j <= right; j++) {
222           sift_down(j, n);
223         }
224       }
225       size_type j = 0;
226       if (right != 0) {
227         j = distinguished_ancestor(right);
228         sift_down(j, n);
229       }
230       if (left != 0) {
231         size_type i = distinguished_ancestor(left);
232         sift_down(i, n);
233         sift_up(i);
234       }
235       sift_up(j);
236     }
237   };
238 }
```

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  #include <cstddef> // std::size_t
8  #include "cartesian-tree.h++"
9  #include "cartesian-tree-node.h++"
10 #include <functional> // std::binary_function
11 #include <iterator> // std::iterator_traits
12 #include "weak-heap-with-buffer.h++"
13 #include <vector>
14
15 namespace cphstl {
16
17   template <typename N, typename C>
18   class indirect_access
19     : public std::binary_function<N*, N*, bool> {
20   protected:
21
22     C less;
```

```
23
24  public:
25
26    explicit indirect_access(const C& f)
27      : less(f) {
28    }
29
30    bool operator()(N* const x, N* const y) const {
31      return less((*x).element(), (*y).element());
32    }
33  };
34
35  template <typename R, typename C,
36            typename N = cphstl::cartesian_tree_node<typename std::iterator_traits<R
                >::value_type> >
37  void adaptive_heapsort(R start, R end, C comparator) {
38    typedef R random_access_iterator;
39    typedef C comparator_type;
40    typedef N node_type;
41    typedef cphstl::indirect_access<N, C> D;
42    typedef std::size_t size_type;
43    typedef std::vector<N> node_pool_type;
44
45    // create the pool of nodes used by the Cartesian tree
46
47    size_type n = end - start;
48    node_pool_type node_pool(n);
49
50    R i = start;
51    N* p = &node_pool[0];
52    while (i != end) {
53      (*p).element() = *i;
54      ++p;
55      ++i;
56    }
57
58    // construct the Cartesian tree
59
60    cphstl::cartesian_tree<node_pool_type, C> tree(node_pool, comparator);
61
62    // sort the elements
63
64    D indirect_comparator(comparator);
65    cphstl::weak_heap<N*, D> heap(indirect_comparator, n);
66    heap.insert(tree.minimum());
67    R output = start;
68    while (heap.size() > 0) {
69      N* q = heap.minimum();
70      *output = (*q).element();
71      ++output;
72      N* c = (*q).first();
73      N* d = (*q).second();
74      if (c != 0) {
75        heap.update_min(c);
76        if (d != 0) {
77          heap.insert(d);
78        }
79      }
80      else {
81        if (d != 0) {
82          heap.update_min(d);
83        }
84        else {
85          heap.extract_min();
86        }
```

```
87        }
88      }
89    }
90  }
```

§ 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
15   template <typename E>
16   class composite_node {
17   public:
18
19     typedef E element_type;
20
21     composite_node* north;
22     composite_node* south;
23     composite_node* east;
24     composite_node* west;
25     E value;
26
27     explicit composite_node() :
28       north(0), south(0), east(0), west(0) {
29     }
30
31     explicit composite_node(E const & v) :
32       north(0), south(0), east(0), west(0), value(v) {
33     }
34
35     E const & element() const {
36       return value;
37     }
38
39     composite_node* first() const {
```

```
40      return south;
41    }
42
43    composite_node* second() const {
44      return north;
45    }
46
47    composite_node* previous() const {
48      return west;
49    }
50
51    composite_node* next() const {
52      return east;
53    }
54
55    composite_node* left() const {
56      return west;
57    }
58
59    composite_node* right() const {
60      return east;
61    }
62
63    E& element() {
64      return value;
65    }
66
67    void element(E const& new_value) {
68      value = new_value;
69    }
70
71    void first(composite_node* v) {
72      south = v;
73    }
74
75    void second(composite_node* v) {
76      north = v;
77    }
78
79    void previous(composite_node* v) {
80      west = v;
81    }
82
83    void next(composite_node* v) {
84      east = v;
85    }
86
87    void left(composite_node* v) {
88      west = v;
89    }
90
91    void right(composite_node* v) {
92      east = v;
93    }
94
95    template <typename C>
96    composite_node* link(composite_node* q, C const& comparator) {
97      composite_node* p = this;
98      if (comparator((*q).element(), (*p).element())) {
99        std::swap(q, p);
100     }
101     composite_node* c = (*p).right();
102     (*q).left(c);
103     (*p).left(0);
104     (*p).right(q);
```

```
105       return p;
106     }
107
108     void swap_roots(composite_node* q) { // must be outside a tree inventory
109       composite_node* p = this;
110       composite_node* a = (*p).right();
111       composite_node* b = (*q).right();
112       (*q).right(a);
113       (*p).right(b);
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 #include <algorithm>
13 #include <cstddef> // std::size_t
14 #include <vector>
15
16 namespace cphstl {
17
18   template <typename N, typename C>
19   class list_based_buffer {
20   public:
21
22     // types
23
24     typedef N node_type;
25     typedef C comparator_type;
26     typedef std::size_t size_type;
27
28   protected:
29
30     // variables
31
32     size_type n;
33     C comparator;
34     N* head;
35
36   public:
37
38     // structors
39
40     explicit list_based_buffer(C const & c = C())
41       : n(0), comparator(c), head(0) {
42     }
43
44     ~list_based_buffer() {
45     }
46
47     // accessors
```

```
48
49      size_type size() const {
50        return n;
51      }
52
53      N* minimum() const {
54        return head;
55      }
56
57      // modifiers
58
59      void insert(N* p) {
60        if (head == 0) {
61          (*p).next(p);
62          head = p;
63        }
64        else {
65          (*p).next((*head).next());
66          (*head).next(p);
67          if (comparator((*p).element(), (*head).element())) {
68            head = p;
69          }
70        }
71        ++n;
72      }
73
74      N* borrow() {
75        --n;
76        if (n == 0) {
77          N* p = head;
78          (*p).next(0);
79          head = 0;
80          return p;
81        }
82        N* p = (*head).next();
83        (*head).next((*p).next());
84        (*p).next(0);
85        return p;
86      }
87
88      N* extract_min() {
89        --n;
90        if (n == 0) {
91          N* p = head;
92          (*p).next(0);
93          head = 0;
94          return p;
95        }
96        N* p = head;
97        N* q = (*p).next();
98        N* minimum = q;
99        N* r = (*q).next();
100       while (r != head) {
101         if (comparator((*r).element(), (*minimum).element())) {
102           minimum = r;
103         }
104         q = r;
105         r = (*r).next();
106       }
107       (*q).next((*p).next());
108       (*p).next(0);
109       head = minimum;
110       return p;
111     }
112   };
```

```
113 }
```

§ 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 #include <climits> // CHAR_BIT
12 #include <cmath> // ilogb
13 #include <cstddef> // std::size_t
14
15 extern int ilogb(double) throw();
16
17 namespace cphstl {
18
19   template<typename N, typename C>
20   class binary_system_tree_inventory {
21   public:
22
23     typedef N node_type;
24     typedef C comparator_type;
25     typedef std::size_t size_type;
26
27   protected:
28
29     static const size_type address_size = CHAR_BIT * sizeof(N*);
30     size_type n;
31     C comparator;
32     N* header[address_size];
33
34   public:
35
36     binary_system_tree_inventory(C const & c = C())
37       : n(0), comparator(c) {
38       for (size_type i = 0; i != __end__(); ++i) {
39         header[i] = 0;
40       }
41     }
42
43     ~binary_system_tree_inventory() {
44     }
45
46     size_type __rank__(N* const p) const {
47       size_type r = 0;
48       while (r != __end__()) {
49         if (header[r] == p) {
50           return r;
51         }
52         ++r;
53       }
54       return __end__();
55     }
56
57     size_type __first__() const {
58       if (n == 0) {
59         return __end__();
60       }
```

```
61        size_type r = 0;
62        size_type mask = n;
63        while (r != __end__()) {
64          if ((mask & 1) == 1) {
65            return r;
66          }
67          ++r;
68          mask >>= 1;
69        }
70        return __end__();
71      }
72
73      size_type __last__() const {
74        if (n == 0) {
75          return __end__();
76        }
77        return size_type(ilogb(n));
78      }
79
80      size_type __end__() const {
81        return address_size;
82      }
83
84      N* __access__(size_type rank) const {
85        return header[rank];
86      }
87
88      size_type __next__(size_type current) const {
89        size_type r = current + 1;
90        size_type mask = n >> r;
91        if (mask == 0) {
92          return __end__();
93        }
94        while (r != __end__()) {
95          if ((mask & 1) == 1) {
96            return r;
97          }
98          ++r;
99          mask >>= 1;
100       }
101       return __end__();
102     }
103
104     void insert(size_type rank, N* p) {
105       N* q = header[rank];
106       n += (1 << rank);
107       while (q != 0) {
108         p = (*p).link(q, comparator);
109         header[rank] = 0;
110         ++rank;
111         q = header[rank];
112       }
113       header[rank] = p;
114     }
115
116     void extract(size_type rank, N* q) {
117       n -= (1 << rank);
118       header[rank] = 0;
119     }
120
121     void update(size_type rank, N* q) {
122       header[rank] = q;
123     }
124   };
125 }
```

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

```
64
65      template <typename T>
66      void refresh(size_type forward_from, T const & tree_inventory) {
67        size_type old_max_rank = 0;
68        if (n != 0) {
69          old_max_rank = ilogb(n);
70        }
71        for (size_type i = forward_from; i <= old_max_rank; ++i) {
72          prefix_minima[i] = 0;
73        }
74        n &= (1 << forward_from) - 1; // unset high bits
75        N* minimum = 0;
76        for (size_type i = 0; i < forward_from; ++i) {
77          if (tree_inventory.__access__(i) != 0) {
78            minimum = prefix_minima[i];
79          }
80        }
81        size_type new_max_rank = tree_inventory.__last__();
82        if (new_max_rank == tree_inventory.__end__()) {
83          return;
84        };
85        for (size_type i = forward_from; i <= new_max_rank; ++i) {
86          N* p = tree_inventory.__access__(i);
87          if (p == 0) {
88            continue;
89          }
90          n += (1 << i); // set bit
91          if (minimum == 0) {
92            prefix_minima[i] = p;
93            minimum = p;
94          }
95          else if (comparator((*p).element(), (*minimum).element())) {
96            prefix_minima[i] = p;
97            minimum = p;
98          }
99          else {
100           prefix_minima[i] = minimum;
101         }
102       }
103     }
104   };
105 }
```

§ 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 # include <algorithm>
17 # include "binary-system-tree-inventory.h++"
18 # include <cmath> // ilogb
19 # include <cstddef> // std::size_t
```

```
20 #include "list-based-buffer.h++"
21 #include "prefix-array.h++"
22 #include <vector>
23
24 extern int ilogb(double) throw();
25
26 namespace cphstl {
27
28   template <typename N, typename C>
29   class weak_queue {
30   public:
31
32     // types
33
34     typedef N node_type;
35     typedef C comparator_type;
36     typedef typename N::element_type element_type;
37     typedef cphstl::list_based_buffer<N, C> buffer_type;
38     typedef cphstl::prefix_array<N, C> prefix_array_type;
39     typedef cphstl::binary_system_tree_inventory<N, C> tree_inventory_type;
40     typedef std::size_t size_type;
41
42   protected:
43
44     // variables
45
46     size_type n;
47     comparator_type comparator;
48     buffer_type buffer;
49     tree_inventory_type tree_inventory;
50     prefix_array_type prefix_array;
51
52   public:
53
54     // structors
55
56     explicit weak_queue(C const & c = C())
57       : n(0), comparator(c), buffer(c), tree_inventory(c), prefix_array(c) {
58     }
59
60     ~weak_queue() {
61     }
62
63     // accessors
64
65     size_type size() const {
66       return n;
67     }
68
69     N* minimum() const {
70       N* p = buffer.minimum();
71       N* q = prefix_array.minimum();
72       if (p == 0) {
73         return q;
74       }
75       if (q == 0) {
76         return p;
77       }
78       if (comparator((*q).element(), (*p).element())) {
79         return q;
80       }
81       return p;
82     }
83
84     // modifiers
```

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

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

Adaptive heapsort using a weak queue

§ 12 *adaptive-heapsort-with-weak-queue.h++*

```
1 /*
```

```
2   Adaptive heapsort using a weak queue
3
4   Authors: Stefan Edelkamp, Jyrki Katajainen © 2010, 2011
5 */
6
7 #include <cstddef> // std::size_t
8 #include <cstdlib>
9 #include "cartesian_tree.h++"
10 #include <cmath> // srand, rand
11 #include "composite_node.h++"
12 #include <functional>
13 #include <iterator> // std::iterator_traits
14 #include "weak_queue_with_buffer.h++"
15 #include <vector>
16
17 namespace cphstl {
18
19   template <typename R, typename C,
20             typename N = cphstl::composite_node<typename std::iterator_traits<R>::
                 value_type> >
21   void adaptive_heapsort(R start, R end, C comparator) {
22     typedef R random_access_iterator;
23     typedef C comparator_type;
24     typedef N node_type;
25     typedef std::size_t size_type;
26     typedef std::vector<N> node_pool_type;
27
28     // create the pool of nodes used by all the data structures
29
30     size_type n = end - start;
31     node_pool_type node_pool(n);
32
33     R i = start;
34     N* p = &node_pool[0];
35     while (i != end) {
36       (*p).element() = *i;
37       ++p;
38       ++i;
39     }
40
41     // construct the Cartesian tree
42
43     cphstl::cartesian_tree<node_pool_type, C> tree(node_pool, comparator);
44
45     // sort the elements
46
47     cphstl::weak_queue<N, C> queue(comparator);
48     queue.insert(tree.minimum());
49     R output = start;
50     while (queue.size() > 0) {
51       N* q = queue.minimum();
52       *output = (*q).element();
53       ++output;
54       N* c = (*q).first();
55       N* d = (*q).second();
56       if (c != 0) {
57         (void) queue.update_min(c);
58         if (d != 0) {
59           queue.insert(d);
60         }
61       }
62       else {
63         if (d != 0) {
64           (void) queue.update_min(d);
65         }
```

```
66          else {
67            (void) queue.extract_min();
68          }
69        }
70      }
71    }
72  }
```

§ 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.c++*

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

§ 18 *comparison-count.c++*

```
1  #include <algorithm> // std::copy, std::equal, std::sort
2  #include "algorithm.i++" // ALGORITHM
3  #include <cstddef> // std::size_t
4  #include <functional> // std::binary_function
5  #include "generator.i++" // GENERATOR
6  #include <iostream>
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 int main(int argc, char *argv[]) {
26   typedef int E;
27   typedef counting_comparator<E> C;
28   typedef std::vector<E> D;
29   typedef std::size_t size_type;
30
31   size_type n = NUMBER;
32   std::cout<< n << " ";
33
34   D data(n);
35   GENERATOR(data);
36
37   D copy(n);
38   std::copy(data.begin(), data.end(), copy.begin());
39
40   comps = 0;
41
42   ALGORITHM(data.begin(), data.end(), C());
43
44   std::cout << (float) comps / n;
45   std::cout << std::endl;
46
47   std::sort(copy.begin(), copy.end(), C());
48   return 0;
49 }
```

Makefile

§ 19 *benchmark.mk*

```
1 CXXFLAGS = -Wall -Wno-unused-local-typedefs -Wno-long-long -std=c++11 -pedantic -
      DNDEBUG -O3
2 CXX = g++
3
4 implementations:= $(wildcard *.i++)
5 algorithms:= $(basename $(implementations))
6 time-tests = $(addsuffix .time, $(algorithms))
7 comp-tests = $(addsuffix .comp, $(algorithms))
8
9 test = 500000 5000000 # 50000000
10 data-type = "random_permutation" # "zero_sequence" "type_one" "type_two"
11
12 $(time-tests): %.time : %.i++
13         @echo $* "running time (s)"
14         @cp $*.i++ algorithm.i++
15         @for d in $(data-type) ; do \
16           echo $$d; \
17           for x in $(test) ; do \
18             $(CXX) $(CXXFLAGS) -DNUMBER=$$x -DGENERATOR=$$d running-time.c++; \
19             ./a.out; \
20             rm -f ./a.out ; \
21           done \
22         done
23         @rm algorithm.i++
24
25 $(comp-tests): %.comp : %.i++
26         @echo $* "# comparisons per element"
```

```
27          @cp $*.i++ algorithm.i++
28          @for d in $(data-type) ; do \
29            echo $$d; \
30            for x in $(test) ; do \
31              $(CXX) $(CXXFLAGS) -DNUMBER=$$x -DGENERATOR=$$d comparison-count.c++;\
32              ./a.out; \
33              rm -f ./a.out ; \
34            done \
35          done
36          @rm algorithm.i++
37
38  clean:
39          @rm -vf *~ a.out core algorithm.i++
```