All-in-one implementation framework for binary heaps: Electronic appendix

Jyrki Katajainen

Department of Computer Science, University of Copenhagen
Universitetsparken 5, 2100 Copenhagen East, Denmark

Abstract. In the paper "All-in-one implementation framework for binary heaps", I described a generic framework for implementing a binary heap. By customizing the type parameters, I used the framework to derive 14 different implementations. Then I benchmarked these implementations against each other and against some of the best competitors. My conclusion is that the paper describes the state of the art: After optimizing the code, the framework had very little overhead which was a central question raised in the paper. In this electronic appendix, and in an accompanying `tar` ball, I release the source code described, discussed, and benchmarked.

My motivation for writing the paper came from some discussions I found on the Internet when googling with the words "pointer-based binary heap". By reading the top-10 answers, it was easy to identify a set of implementation alternatives and potential problems with them:

implicit structure: An array of values; easy to implement and space efficient; can have a catch in dynamization

referent structure: An array of pointers to values; still easy to get access to the neighbours of a node

linked structure: A binary tree of nodes, each storing a value and pointers to nodes; challenging to get correct; how to keep track of the last leaf; how many pointers to use; should there be other information at the nodes.

In addition, several metrics how the efficiency should be measured were specified: worst-case running time, average-case running time, amortized running time, and space consumption. However, very few facts were given how well different alternatives perform according to these metrics. By reading the paper and running the code, you can determine what is the best alternative for your application.

Keywords. Software libraries, frameworks, priority queues, binary heaps

CPH STL Report 2015-2, August 2015.

Release date

2015-08-19

Acknowledgement

I thank Claus Jensen for allowing me to release his implementation of the referent binary heap.

Directory structure

The directory structure follows closely the packages developed. The listing below gives the directories in alphabetical order. Later in this report the packages are described in the order they appear in the paper.

Aids/	Implicit/	Tuned-implicit/
benchmark.mk	Inverse/	Tuned-inverse/
Common/	Jensen/	Tuned-linked/
Drivers/	Linked/	Tuned-referent
Edelkamp-Katajainen/	Referent/	Williams/
Goodrich-et-al/	Std/	

The directory Common is supposed to contain all files derived from the CPH STL, in particular, the files related to different versions of dynamic arrays, but, because this is work in progress, the files are not included. Early versions of our dynamic arrays appear in CPH STL Report 2012-3, but they have been corrected and improved since then.

The programs in the directory Edelkamp-Katajainen were taken directly from CPH STL Report 2009-7. Only one compilation error was corrected, after which the code worked. Therefore, these files are not included here. The configuration file binary_heap.i++ in that directory shows which files are needed from the old package.

The directories Aids and Drivers contain files that are used by the makefile benchmark.mk to run the benchmarks. These files are given at the end of this report together with that makefile. You will also find the details how to do profiling from the makefile.

In my original, the directories Goodrich-et-al and Williams contain programs, for which I do not have the copyright, so these files are not reproduced here.

In general, a file with an extension .h++ contains a definition of a class template, a file with an extension .i++ describes how such a class template is configured, and a file with an extension .c++ is a program that can be used to run an experiment. Observe that the driver programs assume that the makefile creates the files needed by each individual experiment. This way the benchmark programs could be made generic.

From the programs all debugging and testing aids have been removed by our build system. If you encounter an error, you are on your own. In the case of an insurmountable problem, you can contact me jyrki@di.ku.dk.

Package overview

Package	Files	Page
Implicit	binary_heap.h++ implicit_nearly_complete_binary_tree.h++ rank_navigator.h++ binary_heap.i++	5
Referent	binary_heap.h++ referent_nearly_complete_binary_tree.h++ rank_navigator.h++ value_encapsulator.h++ node_factory.h++ binary_heap.i++	16
Inverse	binary_heap.h++ inverse_nearly_complete_binary_tree.h++ inverse_link_navigator.h++ inverse_tree_node.h++ node_factory.h++ binary_heap.i++	22
Linked	binary_heap.h++ linked_nearly_complete_binary_tree.h++ link_navigator.h++ binary_tree_node.h++ compact_binary_tree_node.h++ node_factory.h++ binary_heap.i++	31
Std	binary_heap.i++	47
Williams	None	47
Jensen	binary_heap.h++ binary_heap.c++ binary_heap.i++	47
Edelkamp-Katajainen	binary_heap.i++	51
Goodrich-et-al	None	52
Tuned-implicit	binary_heap.h++ implicit_nearly_complete_binary_tree.h++ rank_navigator.h++ binary_heap.i++	52
Tuned-referent	binary_heap.h++ referent_nearly_complete_binary_tree.h++ rank_navigator.h++ value_encapsulator.h++ node_factory.h++ binary_heap.i++	64

Continued on next page

Package overview—Continued from previous page

Implicit package

Implicit/binary_heap.h++

```
1 /*
2   Binary-heap class template for all-in-one implementation framework
3
4   Version: Unoptimized
5
6   Author: Jyrki Katajainen © 2014, 2015
7 */
8
9 #include <cstddef> // std::size_t
```

```
10 #include <utility> // std::move
11
12 namespace cphstl {
13
14   template <typename V, typename T, typename C>
15   class binary_heap {
16   public:
17
18     // types
19
20     using value_type = V;
21     using tree_type = T;
22     using comparator_type = C;
23     using size_type = std::size_t;
24
25   private:
26
27     using M = typename T::const_navigator;
28     using N = typename T::navigator;
29
30     // constants
31
32     M none;
33
34     // variables
35
36     T tree;
37     C less;
38
39     template <typename X>
40     void ignore_unused_variable_warning(X const&) {
41     }
42
43   public:
44
45     // structors
46
47     explicit binary_heap(C const& c = C())
48       : tree(), less(c) {
49     }
50
51     template <typename S>
52     binary_heap(C const& c, S const& s)
53       : tree(), less(c) {
54       try {
55         for (size_type i = 0; i != s.size(); ++i) {
56           (void) tree.expand();
57         }
58       }
59       catch (...) {
60         clear();
61         throw;
```

```
62        }
63        if (tree.size() ≠ 0) {
64          auto final = build(tree.root(), s.begin());
65          ignore_unused_variable_warning(final);
66        }
67      }
68
69      ~binary_heap() {
70      }
71
72      binary_heap(binary_heap const& other)
73        : tree(other.tree), less(other.less) {
74      }
75
76      binary_heap(binary_heap&& other)
77        : tree(std::move(other.tree)), less(other.less) {
78      }
79
80      binary_heap& operator=(binary_heap const& other) {
81        less = other.less;
82        binary_heap tmp(other);
83        clear();
84        tree.swap(tmp.tree);
85        return *this;
86      }
87
88      binary_heap& operator=(binary_heap&& other) {
89        less = other.less;
90        binary_heap tmp(std::move(other));
91        clear();
92        tree.swap(tmp.tree);
93        return *this;
94      }
95
96      // accessors
97
98      size_type size() const {
99        return tree.size();
100     }
101
102     V const& top() const {
103       return *tree.root();
104     }
105
106     // modifiers
107
108     void push(V const& in) {
109       N hole = tree.expand();
110       siftup(hole, in);
111       *hole = in;
112     }
113
```

```
114     void push(V&& in) {
115       N hole = tree.expand();
116       siftup(hole, in);
117       *hole = std::move(in);
118     }
119
120     void pop() {
121       if (tree.size() == 1) {
122         tree.contract();
123       }
124       else {
125         N hole = tree.root();
126         N leaf = tree.last();
127         siftdown(hole, *leaf);
128         hole.swap(leaf);
129         tree.contract();
130       }
131     }
132
133     void pop(V& out) {
134       out = std::move(*tree.root());
135       pop();
136     }
137
138     void clear() {
139       while (tree.size() != 0) {
140         tree.contract();
141       }
142     }
143
144   private:
145
146     void siftup(N& hole, V const& in) {
147       while (hole != tree.root()) {
148         N p = hole.parent();
149         if (not less(*p, in)) {
150           break;
151         }
152         hole.slide(p);
153       }
154     }
155
156     void siftdown(N& hole, V const& in) {
157       N p = hole.left();
158       while (p != none) {
159         N q = hole.right();
160         if (q != none and less(*p, *q)) {
161           p = q;
162         }
163         if (not less(in, *p)) {
164           break;
165         }
```

```
166         hole.slide(p);
167         p = hole.left();
168       }
169     }
170
171     template <typename I>
172     I build(N p, I t) {
173       if (p == none) {
174         return t;
175       }
176       t = build(p.left(), t);
177       t = build(p.right(), t);
178       siftdown(p, *t);
179       *p = *t;
180       return ++t;
181     }
182   };
183 }
```

Implicit/implicit_nearly_complete_binary_tree.h++

```
1  /*
2    An implicit nearly-complete binary tree needed in the
3    implementation framework for binary heaps
4
5    Version: Unoptimized
6
7    Author: Jyrki Katajainen @ 2012, 2013, 2015
8  */
9
10 #ifndef __CPHSTL_IMPLICIT_NEARLY_COMPLETE_BINARY_TREE__
11 #define __CPHSTL_IMPLICIT_NEARLY_COMPLETE_BINARY_TREE__
12
13 #include <cstddef> // std::size_t
14 #include <memory> // std::allocator
15 #include "rank_navigator.h++"
16 #include <vector> // std::vector
17 #include <utility> // std::move
18
19 namespace cphstl {
20
21   template <typename V, typename S = std::vector<V, std::allocator<V
       >>>
22   class implicit_nearly_complete_binary_tree {
23   public:
24
25     using value_type = V;
26     using sequence_type = S;
27     using allocator_type = typename S::allocator_type;
28     using reference = V&;
29     using const_reference = V const&;
```

```
30    using size_type = std::size_t;
31    using self_type = cphstl::implicit_nearly_complete_binary_tree<V,
          S>;
32    using navigator = cphstl::rank_navigator<self_type>;
33    using const_navigator = cphstl::rank_navigator<self_type const>;
34
35  private:
36
37    friend navigator;
38    friend const_navigator;
39
40    S sequence;
41
42    const_reference operator[](size_type i) const {
43      return sequence[i];
44    }
45
46    reference operator[](size_type i) {
47      return sequence[i];
48    }
49
50  public:
51
52    explicit implicit_nearly_complete_binary_tree()
53      : sequence() {
54    }
55
56    implicit_nearly_complete_binary_tree(S const& s)
57      : sequence(s) {
58    }
59
60    template <typename X>
61    implicit_nearly_complete_binary_tree(X const& s)
62      : sequence(s.get_allocator()) {
63      sequence.reserve(s.size());
64      try {
65        for (V x : s) {
66          sequence.push_back(x);
67        }
68      }
69      catch (...) {
70        sequence.clear();
71        throw;
72      }
73    }
74
75    implicit_nearly_complete_binary_tree(S&& s)
76      : sequence(std::move(s)) {
77    }
78
79    implicit_nearly_complete_binary_tree(self_type const& other)
80      : sequence(other.sequence) {
```

```
81      }
82
83      implicit_nearly_complete_binary_tree(self_type&& other)
84        : sequence(std::move(other.sequence)) {
85      }
86
87      ~implicit_nearly_complete_binary_tree() {
88        clear();
89      }
90
91      self_type& operator=(self_type const& other) {
92        self_type tmp(other);
93        clear();
94        swap(tmp);
95        return *this;
96      }
97
98      self_type& operator=(self_type&& other) {
99        self_type tmp(std::move(other));
100       clear();
101       swap(tmp);
102       return *this;
103     }
104
105     allocator_type get_allocator() const {
106       return sequence.get_allocator();
107     }
108
109     size_type size() const {
110       return sequence.size();
111     }
112
113     const_navigator root() const {
114       return const_navigator(this, 0);
115     }
116
117     navigator root() {
118       return navigator(this, 0);
119     }
120
121     const_navigator last() const {
122       return const_navigator(this, size() - 1);
123     }
124
125     navigator last() {
126       return navigator(this, size() - 1);
127     }
128
129     navigator expand() {
130       sequence.push_back(std::move(V()));
131       return last();
132     }
```

```
133
134     void contract() {
135       sequence.pop_back();
136     }
137
138     void swap(self_type& other) {
139       sequence.swap(other.sequence);
140     }
141
142     void clear() {
143       sequence.clear();
144     }
145   };
146 }
147
148 #endif
```

Implicit/rank_navigator.h++

```
1  /*
2    A rank navigator encapsulates a position of a value by storing a
3    (pointer, rank) pair; the pointer refers to the container—the
4    owner—that contains the value referred to and the rank is the
5    index of that value within the container.
6
7    Version: Unoptimized
8
9    Authors: Jyrki Katajainen, Andreas Milton Maniotis, Bo Simonsen
10   © 2008, 2012, 2013, 2015
11 */
12
13 #ifndef __CPHSTL_RANK_NAVIGATOR__
14 #define __CPHSTL_RANK_NAVIGATOR__
15
16 #include <type_traits> // std::conditional, std::is_const, std::
       remove_const
17 #include <utility> // std::move, std::swap
18
19 namespace cphstl {
20
21   template <typename T>
22   class rank_navigator {
23
24   public:
25
26     // type aliases
27
28     using owner_type = T;
29
30   private:
31
```

```
32      using T_bar = typename std::conditional<std::is_const<T>::value,
            typename std::remove_const<T>::type, T const>::type;
33      using R = rank_navigator<T>;
34      using R_bar = rank_navigator<T_bar>;
35      using size_type = typename T::size_type;
36      using V = typename T::value_type;
37      using V_check = typename std::conditional<std::is_const<T>::value
            , V const, V>::type;
38      using pointer = V_check*;
39      using reference = V_check&;
40
41      // friends
42
43      friend T;
44      friend R_bar;
45
46      // constants
47
48      static size_type const outside = -1;
49
50      // variables
51
52      T* owner_pointer;
53      size_type rank;
54
55      // parameterized constructor
56
57      rank_navigator(owner_type* p, size_type offset)
58        : owner_pointer(p), rank(offset) {
59      }
60
61    public:
62
63      // default constructor
64
65      rank_navigator()
66        : owner_pointer(nullptr), rank(outside) {
67      }
68
69      // copy constructors
70
71      // Generated by the compiler if needed
72      // rank_navigator(const rank_navigator&) = default;
73
74      rank_navigator(rank_navigator<typename std::remove_const<T>::type
            > const& x)
75        : owner_pointer(x.owner_pointer), rank(x.rank) {
76      }
77
78      // assignments
79
80      // Generated by the compiler if needed
```

```
81    // rank_navigator& operator=(rank_navigator const&) = default;
82
83    rank_navigator& operator=(rank_navigator<typename std::
          remove_const<T>::type> const& x) {
84      owner_pointer = x.owner_pointer;
85      rank = x.rank;
86      return *this;
87    }
88
89    // destructor
90
91    ~rank_navigator() {
92    };
93
94    // operator*
95
96    reference operator*() const {
97      return (*owner_pointer)[rank];
98    }
99
100   // operator->
101
102   pointer operator->() const {
103     return &(*owner_pointer)[rank];
104   }
105
106   // left
107
108   R left() const {
109     size_type child = 2 * rank + 1;
110     if (child >= (*owner_pointer).size()) {
111       return R();
112     }
113     return R(owner_pointer, child);
114   }
115
116   // right
117
118   R right() const {
119     size_type child = 2 * rank + 2;
120     if (child >= (*owner_pointer).size()) {
121       return R();
122     }
123     return R(owner_pointer, child);
124   }
125
126   // parent
127
128   R parent() const {
129     if (rank == 0) {
130       return R();
131     }
```

```
132       return R(owner_pointer, (rank - 1) / 2);
133     }
134
135     // slide
136
137     void slide(R& neighbour) {
138       swap(neighbour);
139     }
140
141     // swap
142
143     void swap(R& other) {
144       **this = std::move(*other);
145       std::swap(rank, other.rank);
146     }
147
148     // operator==
149
150     bool operator==(R const& other) const {
151       return (rank == other.rank) and (owner_pointer == other.
              owner_pointer);
152     }
153
154     bool operator==(R_bar const& other) const {
155       return (rank == other.rank) and (owner_pointer == other.
              owner_pointer);
156     }
157
158     // operator!=
159
160     bool operator!=(R const& other) const {
161       return not (*this == other);
162     }
163
164     bool operator!=(R_bar const& other) const {
165       return not (*this == other);
166     }
167   };
168 }
169
170 #endif
```

Implicit/binary_heap.i++

```
1 #include "element.i++" // V
2 #include "comparator.i++" // C
3 #include "experiment.i++"  // S
4 #include "implicit_nearly_complete_binary_tree.h++"
5 #include "binary_heap.h++" // cphstl::binary_heap
6
7 using T = cphstl::implicit_nearly_complete_binary_tree<V, S>;
```

```
8 using Q = cphstl::binary_heap<V, T, C>;
```

Referent package

Referent/binary_heap.h++

This file is a symbolic link to Implicit/binary_heap.h++.

Referent/referent_nearly_complete_binary_tree.h++

```
1 /*
2   A referent nearly-complete binary tree needed in the implementation
3   framework for binary heaps
4
5   Version: Unoptimized
6
7   Author: Jyrki Katajainen @ 2012, 2013, 2015
8 */
9
10 #ifndef __CPHSTL_REFERENT_NEARLY_COMPLETE_BINARY_TREE__
11 #define __CPHSTL_REFERENT_NEARLY_COMPLETE_BINARY_TREE__
12
13 #include <cstddef> // std::size_t
14 #include <memory> // std::allocator
15 #include "node_factory.h++"
16 #include "rank_navigator.h++"
17 #include "value_encapsulator.h++"
18 #include <vector> // std::vector
19 #include <utility> // std::move
20
21 namespace cphstl {
22
23   template <
24     typename V,
25     typename S = std::vector<V, std::allocator<V>>,
26     typename N = cphstl::value_encapsulator<V>,
27     typename F = cphstl::node_factory<N, typename S::allocator_type>
28   >
29   class referent_nearly_complete_binary_tree {
30   public:
31
32     using value_type = V;
33     using sequence_type = S;
34     using node_type = N;
35     using factory_type = F;
36     using allocator_type = typename S::allocator_type;
37     using reference = V&;
38     using const_reference = V const&;
39     using size_type = std::size_t;
```

```
40    using self_type = cphstl::referent_nearly_complete_binary_tree<V,
          S, N, F>;
41    using navigator = cphstl::rank_navigator<self_type>;
42    using const_navigator = cphstl::rank_navigator<self_type const>;
43
44  private:
45
46    friend navigator;
47    friend const_navigator;
48
49    template <typename T, typename U, typename Some>
50    struct rebind;
51
52    template <typename T, typename U, template <typename...> class
          Name, typename... Args>
53    struct rebind<T, U, Name<Args...>> {
54      typedef Name<T, U> other;
55    };
56
57    using A = typename S::allocator_type;
58    using A_prime = typename A::template rebind<N>::other;
59    using A_prime_prime = typename A::template rebind<N*>::other;
60    using S_prime_prime = typename rebind<N*, A_prime_prime, S>::
          other;
61    using F_prime = typename rebind<N, A_prime, F>::other;
62
63    S_prime_prime sequence;
64    F_prime factory;
65
66    const_reference operator[](size_type i) const {
67      return (*sequence[i]).value();
68    }
69
70    reference operator[](size_type i) {
71      return (*sequence[i]).value();
72    }
73
74  public:
75
76    explicit referent_nearly_complete_binary_tree(F const& f = F())
77      : sequence(f.get_allocator()), factory(f) {
78    }
79
80    template <typename X>
81    referent_nearly_complete_binary_tree(X const& s, F const& f = F()
          )
82      : sequence(s.get_allocator()), factory(f) {
83      sequence.reserve(s.size());
84      try {
85        for (V x : s) {
86          sequence.push_back(factory.create(x));
87        }
```

```
88        }
89        catch (...) {
90          clear();
91          throw;
92        }
93      }
94
95      referent_nearly_complete_binary_tree(self_type&& other)
96        : sequence(other.get_allocator()), factory(other.factory) {
97        swap(other);
98      }
99
100     ~referent_nearly_complete_binary_tree() {
101       clear();
102     }
103
104     self_type& operator=(self_type const& other) {
105       self_type tmp(other);
106       clear();
107       swap(tmp);
108       return *this;
109     }
110
111     self_type& operator=(self_type&& other) {
112       self_type tmp(std::move(other));
113       clear();
114       swap(tmp);
115       return *this;
116     }
117
118     allocator_type get_allocator() const {
119       return sequence.get_allocator();
120     }
121
122     size_type size() const {
123       return sequence.size();
124     }
125
126     const_navigator root() const {
127       return const_navigator(this, 0);
128     }
129
130     navigator root() {
131       return navigator(this, 0);
132     }
133
134     const_navigator last() const {
135       return const_navigator(this, size() - 1);
136     }
137
138     navigator last() {
139       return navigator(this, size() - 1);
```

```
140     }
141
142     navigator expand() {
143       N* p = factory.create();
144       sequence.push_back(p);
145       return last();
146     }
147
148     void contract() {
149       size_type ℓ = size() - 1;
150       N* p = sequence[ℓ];
151       factory.destroy(p);
152       sequence.pop_back();
153     }
154
155     void swap(self_type& other) {
156       sequence.swap(other.sequence);
157       factory.swap(other.factory);
158     }
159
160     void clear() {
161       for (N* p : sequence) {
162         factory.destroy(p);
163       }
164       sequence.clear();
165     }
166   };
167 }
168
169 #endif
```

Referent/rank_navigator.h++

This file is a symbolic link to Implicit/rank_navigator.h++.

Referent/value_encapsulator.h++

```
1  /*
2    An encapsulator encapsulates a value, nothing else.
3
4    Authors: Jyrki Katajainen, Bo Simonsen © 2008, 2009, 2010, 2015
5  */
6
7  #ifndef __CPHSTL_VALUE_ENCAPSULATOR__
8  #define __CPHSTL_VALUE_ENCAPSULATOR__
9
10 #include <utility> // std::move
11
12 namespace cphstl {
13
```

```
14    template <typename V>
15    class value_encapsulator {
16    public:
17
18      using value_type = V;
19
20      value_encapsulator()
21        : element() {
22      }
23
24      value_encapsulator(V const& v)
25        : element(v) {
26      }
27
28      value_encapsulator(V&& v)
29        : element(std::move(v)) {
30      }
31
32      V const& value() const {
33        return element;
34      }
35
36      V& value() {
37        return element;
38      }
39
40    private:
41
42      V element;
43    };
44  }
45
46  #endif
```

Referent/node_factory.h++

```
1   #ifndef __CPHSTL_NODE_FACTORY__
2   #define __CPHSTL_NODE_FACTORY__
3
4   /*
5     This factory can be used to create and destroy any kinds of nodes
6
7     Authors: Jyrki Katajainen, Bo Simonsen © 2008, 2010, 2015
8   */
9
10  #include <memory> // std::allocator
11  #include <utility> // std::forward
12
13  namespace cphstl {
14
15    template <typename N, typename A = std::allocator<N>>
```

```
16    class node_factory {
17    public:
18
19      using node_type = N;
20      using allocator_type = A;
21      using value_type = typename N::value_type;
22
23    private:
24
25      using V = value_type;
26      using A_prime = typename A::template rebind<N>::other;
27
28      A_prime node_allocator;
29
30    public:
31
32      node_factory(A const& a = A())
33        : node_allocator(a) {
34      }
35
36      template <typename A_prime_prime>
37      node_factory(node_factory<N, A_prime_prime> const& other)
38        : node_allocator(other.get_allocator()) {
39      }
40
41      A get_allocator() const {
42        return A(node_allocator);
43      }
44
45      template <typename... Args>
46      N* create(Args&&... args) {
47        N* p = node_allocator.allocate(1);
48        try {
49          new (p) N(std::forward<Args>(args)...);
50        }
51        catch (...) {
52          node_allocator.deallocate(p, 1);
53          throw;
54        }
55        return p;
56      }
57
58      void destroy(N* p) {
59        (*p).~N();
60        node_allocator.deallocate(p, 1);
61      }
62
63      void swap(node_factory& other) {
64      }
65    };
66  }
67
```

```
68 #endif
```

Referent/binary_heap.i++

```
1  #include "element.i++" // V
2  #include "comparator.i++" // C
3  #include "experiment.i++"  // S
4
5  #include "node_factory.h++" // cphstl::factory
6  #include "referent_nearly_complete_binary_tree.h++"
7  #include "value_encapsulator.h++" // cphstl::encapsulator
8  #include "binary_heap.h++" // cphstl::binary_heap
9
10 using E = cphstl::value_encapsulator<V>;
11 using F = cphstl::node_factory<E, A>;
12 using T = cphstl::referent_nearly_complete_binary_tree<V, S, E, F>;
13 using Q = cphstl::binary_heap<V, T, C>;
```

Inverse package

Inverse/binary_heap.h++

This file is a symbolic link to Implicit/binary_heap.h++.

Inverse/inverse_nearly_complete_binary_tree.h++

```
1  /*
2    An inverse nearly-complete binary tree needed in the implementation
3    framework for binary heaps
4
5    Version: Unoptimized
6
7    Author: Jyrki Katajainen @ 2012, 2013, 2015
8  */
9
10 #ifndef __CPHSTL_INVERSE_NEARLY_COMPLETE_BINARY_TREE__
11 #define __CPHSTL_INVERSE_NEARLY_COMPLETE_BINARY_TREE__
12
13 #include <cstddef> // std::size_t
14 #include "inverse_link_navigator.h++"
15 #include "inverse_tree_node.h++"
16 #include <memory> // std::allocator
17 #include "node_factory.h++"
18 #include <vector> // std::vector
19 #include <utility> // std::move
20
21 namespace cphstl {
22
```

```
23  template <
24    typename V,
25    typename S = std::vector<V, std::allocator<V>>,
26    typename N = cphstl::inverse_tree_node<V>,
27    typename F = cphstl::node_factory<N, typename S::allocator_type>
28  >
29  class inverse_nearly_complete_binary_tree {
30  public:
31
32    using value_type = V;
33    using sequence_type = S;
34    using node_type = N;
35    using factory_type = F;
36    using allocator_type = typename S::allocator_type;
37    using reference = V&;
38    using const_reference = V const&;
39    using size_type = std::size_t;
40    using self_type = cphstl::inverse_nearly_complete_binary_tree<V,
          S, N, F>;
41    using navigator = cphstl::inverse_link_navigator<self_type>;
42    using const_navigator = cphstl::inverse_link_navigator<self_type
          const>;
43
44  private:
45
46    friend navigator;
47    friend const_navigator;
48
49    template <typename T, typename U, typename Some>
50    struct rebind;
51
52    template <typename T, typename U, template <typename...> class
          Name, typename... Args>
53    struct rebind<T, U, Name<Args...>> {
54      typedef Name<T, U> other;
55    };
56
57    using A = typename S::allocator_type;
58    using A_prime = typename A::template rebind<N>::other;
59    using A_prime_prime = typename A::template rebind<N*>::other;
60    using S_prime_prime = typename rebind<N*, A_prime_prime, S>::
          other;
61    using F_prime = typename rebind<N, A_prime, F>::other;
62
63    S_prime_prime sequence;
64    F_prime factory;
65
66    N* const& operator[](size_type i) const {
67      return sequence[i];
68    }
69
70    N*& operator[](size_type i) {
```

```
71      return sequence[i];
72    }
73
74  public:
75
76    explicit inverse_nearly_complete_binary_tree(F const& f = F())
77      : sequence(f.get_allocator()), factory(f) {
78    }
79
80    template <typename X>
81    inverse_nearly_complete_binary_tree(X const& s, F const& f = F())
82      : sequence(s.get_allocator()), factory(f) {
83      sequence.reserve(s.size());
84      size_type i = 0;
85      try {
86        for (V x : s) {
87          sequence.push_back(factory.create(x, i));
88          i += 1;
89        }
90      }
91      catch (...) {
92        while (i != 0) {
93          --i;
94          factory.destroy(sequence[i]);
95        }
96        sequence.clear();
97        throw;
98      }
99    }
100
101   inverse_nearly_complete_binary_tree(self_type&& other)
102     : sequence(other.get_allocator()), factory(other.factory) {
103     swap(other);
104   }
105
106   ~inverse_nearly_complete_binary_tree() {
107     clear();
108   }
109
110   self_type& operator=(self_type const& other) {
111     self_type tmp(other);
112     clear();
113     swap(tmp);
114     return *this;
115   }
116
117   self_type& operator=(self_type&& other) {
118     self_type tmp(std::move(other));
119     clear();
120     swap(tmp);
121     return *this;
122   }
```

```
123
124     allocator_type get_allocator() const {
125       return sequence.get_allocator();
126     }
127
128     size_type size() const {
129       return sequence.size();
130     }
131
132     const_navigator root() const {
133       return const_navigator(this, sequence[0]);
134     }
135
136     navigator root() {
137       return navigator(this, sequence[0]);
138     }
139
140     const_navigator last() const {
141       return const_navigator(this, sequence[size() - 1]);
142     }
143
144     navigator last() {
145       return navigator(this, sequence[size() - 1]);
146     }
147
148     navigator expand() {
149       size_type past = size();
150       N* p = factory.create(std::move(V()), past);
151       sequence.push_back(p);
152       return navigator(this, p);
153     }
154
155     void contract() {
156       size_type ℓ = size() - 1;
157       N* p = sequence[ℓ];
158       factory.destroy(p);
159       sequence.pop_back();
160     }
161
162     void swap(self_type& other) {
163       sequence.swap(other.sequence);
164       factory.swap(other.factory);
165     }
166
167     void clear() {
168       for (N* p : sequence) {
169         factory.destroy(p);
170       }
171       sequence.clear();
172     }
173   };
174 }
```

```
175
176 #endif
```

Inverse/inverse_link_navigator.h++

```
1  /*
2    An inverse link navigator encapsulates a position of a value by
3    storing a pair of pointers, one to its owner and another to the
4    node where the value is stored
5
6    Version: Unoptimized
7
8    Author: Jyrki Katajainen © 2015
9  */
10
11 #ifndef __CPHSTL_INVERSE_LINK_NAVIGATOR__
12 #define __CPHSTL_INVERSE_LINK_NAVIGATOR__
13
14 #include <limits> // std::numeric_limits
15 #include <type_traits> // std::conditional, std::is_const, std::
       remove_const
16
17 namespace cphstl {
18
19   template <typename T>
20   class inverse_link_navigator {
21
22   public:
23
24     // type aliases
25
26     using owner_type = T;
27
28   private:
29
30     using T_bar = typename std::conditional<std::is_const<T>::value,
          typename std::remove_const<T>::type, T const>::type;
31     using L = inverse_link_navigator<T>;
32     using L_bar = inverse_link_navigator<T_bar>;
33     using size_type = typename T::size_type;
34     using V = typename T::value_type;
35     using N = typename T::node_type;
36     using V_check = typename std::conditional<std::is_const<T>::value
           , V const, V>::type;
37     using pointer = V_check*;
38     using reference = V_check&;
39
40     // friends
41
42     friend T;
43     friend L_bar;
```

```
44
45      // variables
46
47      T* owner_pointer;
48      N* p;
49
50      // parameterized constructor
51
52      inverse_link_navigator(T* p, N* x)
53        : owner_pointer(p), p(x) {
54      }
55
56    public:
57
58      // default constructor
59
60      inverse_link_navigator()
61        : owner_pointer(nullptr), p(nullptr) {
62      }
63
64      // copy constructors
65
66      // Generated by the compiler if needed
67      // inverse_link_navigator(const inverse_link_navigator&) =
            default;
68
69      inverse_link_navigator(inverse_link_navigator<typename std::
            remove_const<T>::type> const& x)
70        : owner_pointer(x.owner_pointer), p(x.p) {
71      }
72
73      // assignments
74
75      // Generated by the compiler if needed
76      // inverse_link_navigator& operator=(inverse_link_navigator const
            &) = default;
77
78      inverse_link_navigator& operator=(inverse_link_navigator<typename
            std::remove_const<T>::type> const& x) {
79        owner_pointer = x.owner_pointer;
80        p = x.p;
81        return *this;
82      }
83
84      // destructor
85
86      ~inverse_link_navigator() {
87      };
88
89      // operator*
90
91      reference operator*() const {
```

```
92        return (*p).value();
93      }
94
95      // operator→
96
97      pointer operator→() const {
98        return &(*p).value();
99      }
100
101     // left
102
103     L left() const {
104       size_type rank = 2 * (*p).rank() + 1;
105       if (rank ≥ (*owner_pointer).size()) {
106         return L();
107       }
108       return L(owner_pointer, (*owner_pointer)[rank]);
109     }
110
111     // right
112
113     L right() const {
114       size_type rank = 2 * (*p).rank() + 2;
115       if (rank ≥ (*owner_pointer).size()) {
116         return L();
117       }
118       return L(owner_pointer, (*owner_pointer)[rank]);
119     }
120
121     // parent
122
123     L parent() const {
124       if ((*p).rank() == 0) {
125         return L();
126       }
127       size_type rank = ((*p).rank() - 1) / 2;
128       return L(owner_pointer, (*owner_pointer)[rank]);
129     }
130
131     // slide
132
133     void slide(L const& neighbour) {
134       swap(neighbour);
135     }
136
137     // swap
138
139     void swap(L const& other) {
140       size_type i = (*p).rank();
141       size_type j = (*other.p).rank();
142       (*p).rank() = j;
143       (*other.p).rank() = i;
```

```
144       (*owner_pointer)[i] = other.p;
145       (*owner_pointer)[j] = p;
146     }
147
148     // operator==
149
150     bool operator==(L const& other) const {
151       return p == other.p;
152     }
153
154     bool operator==(L_bar const& other) const {
155       return p == other.p;
156     }
157
158     // operator!=
159
160     bool operator!=(L const& other) const {
161       return not (*this == other);
162     }
163
164     bool operator!=(L_bar const& other) const {
165       return not (*this == other);
166     }
167   };
168 }
169
170 #endif
```

Inverse/inverse_tree_node.h++

```
1  /*
2    An inverse-tree node encapsulates a value and a rank
3
4    Author: Jyrki Katajainen © 2015
5  */
6
7  #ifndef __CPHSTL_INVERSE_TREE_NODE__
8  #define __CPHSTL_INVERSE_TREE_NODE__
9
10 #include <cstddef> // std::size_t
11 #include <utility> // std::move
12
13 namespace cphstl {
14
15   template <typename V>
16   class inverse_tree_node {
17   public:
18
19     using value_type = V;
20     using size_type = std::size_t;
21
```

```
22      inverse_tree_node()
23        : element(), index(-1) {
24      }
25
26      inverse_tree_node(V const& v, size_type i)
27        : element(v), index(i) {
28      }
29
30      inverse_tree_node(V&& v, size_type i)
31        : element(std::move(v)), index(i) {
32      }
33
34      V const& value() const {
35        return element;
36      }
37
38      size_type rank() const {
39        return index;
40      }
41
42      V& value() {
43        return element;
44      }
45
46      size_type& rank() {
47        return index;
48      }
49
50    private:
51
52      V element;
53      size_type index;
54    };
55  }
56
57  #endif
```

Inverse/node_factory.h++

This file is a symbolic link to `Referent/node_factory.h++`.

Inverse/binary_heap.i++

```
1  #include "element.i++" // V
2  #include "comparator.i++" // C
3  #include "experiment.i++"  // S
4
5  #include "binary_heap.h++"
6  #include "inverse_nearly_complete_binary_tree.h++"
7  #include "inverse_tree_node.h++"
```

```
#include "node_factory.h++"

using N = cphstl::inverse_tree_node<V>;
using F = cphstl::node_factory<N, A>;
using T = cphstl::inverse_nearly_complete_binary_tree<V, S, N, F>;
using Q = cphstl::binary_heap<V, T, C>;
```

Linked package

Linked/binary_heap.h++

This file is a symbolic link to Implicit/binary_heap.h++.

Linked/linked_nearly_complete_binary_tree.h++

```
/*
  A linked nearly-complete binary tree needed in the implementation
  framework for binary heaps

  Version: Unoptimized

  Author: Jyrki Katajainen @ 2012, 2013, 2015
*/

#ifndef __CPHSTL_LINKED_NEARLY_COMPLETE_BINARY_TREE__
#define __CPHSTL_LINKED_NEARLY_COMPLETE_BINARY_TREE__

#include "binary_tree_node.h++"
#include <cmath> // ilogb
#include <cstddef> // std::size_t
#include "link_navigator.h++"
#include <memory> // std::allocator
#include "node_factory.h++"
#include <utility> // std::swap, std::move

namespace cphstl {

  template <
    typename V,
    typename N = cphstl::binary_tree_node<V>,
    typename F = cphstl::node_factory<N, std::allocator<N>>
  >
  class linked_nearly_complete_binary_tree {
  public:

    using value_type = V;
    using node_type = N;
    using factory_type = F;
    using allocator_type = typename F::allocator_type;
```

```
35      using reference = V&;
36      using const_reference = V const&;
37      using size_type = std::size_t;
38      using self_type = cphstl::linked_nearly_complete_binary_tree<V, N
            , F>;
39      using navigator = cphstl::link_navigator<self_type>;
40      using const_navigator = cphstl::link_navigator<self_type const>;
41
42    private:
43
44      friend navigator;
45      friend const_navigator;
46
47      template <typename T, typename U, typename Some>
48      struct rebind;
49
50      template <typename T, typename U, template <typename...> class
            Name, typename... Args>
51      struct rebind<T, U, Name<Args...>> {
52        typedef Name<T, U> other;
53      };
54
55      using A = typename F::allocator_type;
56      using A_prime = typename A::template rebind<N>::other;
57      using F_prime = typename rebind<N, A_prime, F>::other;
58
59      size_type n;
60      N* root_access;
61      N* leaf_access;
62      F_prime factory;
63
64      template <typename M>
65      N* copy(M t) {
66        if (t == M()) { // M() == none
67          return nullptr;
68        }
69        N* q = copy(t.left());
70        N* r = copy(t.right());
71        N* p = factory.create(*t);
72        (*p).set_left(q);
73        (*p).set_right(r);
74        return p;
75      }
76
77    public:
78
79      explicit linked_nearly_complete_binary_tree(F const& f = F())
80        : n(0), root_access(nullptr), leaf_access(nullptr), factory(f)
              {
81        root_access = factory.create();
82        leaf_access = factory.create();
83        (*leaf_access).set_left(leaf_access);
```

```
84        (*leaf_access).set_right(nullptr);
85        (*root_access).set_left(leaf_access);
86        (*root_access).set_right(root_access);
87      }
88
89      template <typename S>
90      linked_nearly_complete_binary_tree(S const& s = S(), F const& f =
            F())
91        : n(0), root_access(nullptr), leaf_access(nullptr), factory(f)
            {
92        root_access = factory.create();
93        leaf_access = factory.create();
94        (*leaf_access).set_left(leaf_access);
95        (*leaf_access).set_right(nullptr);
96        (*root_access).set_left(leaf_access);
97        (*root_access).set_right(root_access);
98        try {
99          for (V x : s) {
100           navigator leaf = expand();
101           *leaf = x;
102         }
103       }
104       catch (...) {
105         clear();
106         throw;
107       }
108     }
109
110     linked_nearly_complete_binary_tree(
            linked_nearly_complete_binary_tree const& other)
111       : n(other.size()), root_access(nullptr), leaf_access(nullptr),
112         factory(other.get_allocator()) {
113       root_access = factory.create();
114       leaf_access = factory.create();
115       (*leaf_access).set_left(leaf_access);
116       (*leaf_access).set_right(nullptr);
117       if (n == 0) {
118         (*root_access).set_left(leaf_access);
119         (*root_access).set_right(root_access);
120         return;
121       }
122       N* p = copy(other.root());
123       (*root_access).set_left(p);
124       (*root_access).set_right(root_access);
125       int const h = ilogb(n);
126       size_type mask = 1 << h;
127       for (int i = 0; i != h; ++i) {
128         mask >>= 1;
129         if ((n bitand mask) == 0) {
130           p = (*p).left();
131         }
132         else {
```

```
133           p = (*p).right();
134         }
135       }
136       (*p).set_left(leaf_access);
137       (*p).set_right(nullptr);
138     }
139
140     linked_nearly_complete_binary_tree(
            linked_nearly_complete_binary_tree&& other)
141       : n(other.size()), root_access(nullptr), leaf_access(nullptr),
142         factory(other.factory) {
143       root_access = factory.create();
144       leaf_access = factory.create();
145       (*leaf_access).set_left(leaf_access);
146       (*leaf_access).set_right(nullptr);
147       (*root_access).set_left(leaf_access);
148       (*root_access).set_right(root_access);
149       swap(other);
150     }
151
152     ~linked_nearly_complete_binary_tree() {
153       clear();
154       factory.destroy(root_access);
155       factory.destroy(leaf_access);
156     }
157
158     template <typename T>
159     linked_nearly_complete_binary_tree& operator=(T const& other) {
160       linked_nearly_complete_binary_tree tmp(other);
161       clear();
162       swap(tmp);
163       return *this;
164     }
165
166     linked_nearly_complete_binary_tree& operator=(
            linked_nearly_complete_binary_tree&& other) {
167       linked_nearly_complete_binary_tree tmp(std::move(other));
168       clear();
169       swap(tmp);
170       return *this;
171     }
172
173     size_type size() const {
174       return n;
175     }
176
177     const_navigator root() const {
178       return const_navigator((*root_access).left());
179     }
180
181     navigator root() {
182       return navigator((*root_access).left());
```

```
183     }
184
185     const_navigator last() const {
186       return const_navigator((*leaf_access).parent());
187     }
188
189     navigator last() {
190       return navigator((*leaf_access).parent());
191     }
192
193     navigator expand() {
194       N* q; // parent of the new last leaf
195       N* p = factory.create();
196       n += 1;
197       int const h = __builtin_ctzl(n);
198       N* const ℓ = (*leaf_access).parent();
199       if (__builtin_popcountl(n) == 1) {
200         q = root_access;
201         if (n == 1) {
202           (*q).set_left(p);
203           (*q).set_right(q);
204         }
205         else {
206           for (int i = 0; i ≠ h; ++i) {
207             q = (*q).left();
208           }
209           (*ℓ).set_left(nullptr);
210           (*q).set_left(p);
211           (*q).set_right(nullptr);
212         }
213       }
214       else if ((n bitand 1) == 0) {
215         q = ℓ;
216         for (int i = 0; i ≠ h + 1; ++i) {
217           q = (*q).parent();
218         }
219         q = (*q).right();
220         for (int i = 1; i < h; ++i) {
221           q = (*q).left();
222         }
223         (*ℓ).set_left(nullptr);
224         (*q).set_left(p);
225         (*q).set_right(nullptr);
226       }
227       else {
228         q = (*ℓ).parent();
229         (*ℓ).set_left(nullptr);
230         (*q).set_right(p);
231       }
232       (*p).set_left(leaf_access);
233       (*p).set_right(nullptr);
234       return navigator(p);
```

```
235     }
236
237     void contract() {
238       N* q; // new last leaf
239       int const h = __builtin_ctzl(n);
240       N* const ℓ = (*leaf_access).parent();
241       if (__builtin_popcountl(n) == 1) {
242         if (n == 1) {
243           q = root_access;
244         }
245         else {
246           q = (*root_access).left();
247           for (int i = 1; i ≠ h; ++i) {
248             q = (*q).right();
249           }
250           N* p = (*ℓ).parent();
251           (*p).set_left(nullptr);
252         }
253       }
254       else if ((n bitand 1) == 0) {
255         q = ℓ;
256         for (int i = 0; i ≠ h + 1; ++i) {
257           q = (*q).parent();
258         }
259         q = (*q).left();
260         for (int i = 0; i ≠ h; ++i) {
261           q = (*q).right();
262         }
263         N* p = (*ℓ).parent();
264         (*p).set_left(nullptr);
265       }
266       else {
267         N* p = (*ℓ).parent();
268         q = (*p).left();
269         (*p).set_right(nullptr);
270       }
271       (*q).set_left(leaf_access);
272       (*q).set_right(nullptr);
273       factory.destroy(ℓ);
274       n -= 1;
275     }
276
277     void swap(linked_nearly_complete_binary_tree& other) {
278       std::swap(n, other.n);
279       std::swap(root_access, other.root_access);
280       std::swap(leaf_access, other.leaf_access);
281       factory.swap(other.factory);
282     }
283
284     void clear() {
285       while (size() ≠ 0) {
286         contract();
```

```
287       }
288     }
289   };
290 }
291
292 #endif
```

Linked/link_navigator.h++

```
1  /*
2    A link navigator encapsulates a position of a value by storing a
3    pointer to the node that stores it; all the functionality is
4    provided by the node
5
6    Version: Unoptimized
7
8    Author: Jyrki Katajainen © 2015
9  */
10
11 #ifndef __CPHSTL_LINK_NAVIGATOR__
12 #define __CPHSTL_LINK_NAVIGATOR__
13
14 #include <limits> // std::numeric_limits
15 #include <sstream> // std::stringstream
16 #include <type_traits> // std::conditional, std::is_const, std::
       remove_const
17
18 namespace cphstl {
19
20   template <typename T>
21   class link_navigator {
22
23   public:
24
25     // type aliases
26
27     using owner_type = T;
28
29   private:
30
31     using T_bar = typename std::conditional<std::is_const<T>::value,
           typename std::remove_const<T>::type, T const>::type;
32     using L = link_navigator<T>;
33     using L_bar = link_navigator<T_bar>;
34     using V = typename T::value_type;
35     using N = typename T::node_type;
36     using V_check = typename std::conditional<std::is_const<T>::value
           , V const, V>::type;
37     using pointer = V_check*;
38     using reference = V_check&;
39
```

```
40      // friends
41
42      friend T;
43      friend L_bar;
44
45      // variables
46
47      N* p;
48
49      // parameterized constructor
50
51      link_navigator(N* q)
52        : p(q) {
53      }
54
55    public:
56
57      // default constructor
58
59      link_navigator()
60        : p(nullptr) {
61      }
62
63      // copy constructors
64
65      // Generated by the compiler if needed
66      // link_navigator(const link_navigator&) = default;
67
68      link_navigator(link_navigator<typename std::remove_const<T>::type
            > const& x)
69        : p(x.p) {
70      }
71
72      // assignments
73
74      // Generated by the compiler if needed
75      // link_navigator& operator=(link_navigator const&) = default;
76
77      link_navigator& operator=(link_navigator<typename std::
            remove_const<T>::type> const& x) {
78        p = x.p;
79        return *this;
80      }
81
82      // destructor
83
84      ~link_navigator() {
85      };
86
87      // operator*
88
89      reference operator*() const {
```

```
90        return (*p).value();
91      }
92
93      // operator→
94
95      pointer operator→() const {
96        return &(*p).value();
97      }
98
99      // left
100
101     L left() const {
102       N* below = (*p).left();
103       bool at_leaf = (below == nullptr) or ((*below).left() == below)
              ;
104       if (at_leaf) {
105         return L();
106       }
107       return L(below);
108     }
109
110     // right
111
112     L right() const {
113       N* below = (*p).right();
114       return L(below);
115     }
116
117     // parent
118
119     L parent() const {
120       N* above = (*p).parent();
121       bool at_root = ((*above).parent() == above);
122       if (at_root) {
123         return L();
124       }
125       return L(above);
126     }
127
128     // slide
129
130     void slide(L const& neighbour) {
131       N* q = neighbour.p;
132       N* o = (*p).parent();
133       N* r = (*q).parent();
134       bool p_on_left = ((*o).left() == p);
135       bool q_on_left = ((*r).left() == q);
136       if (o == q) {
137         if (q_on_left) {
138           if (p_on_left) {
139             left_left_case(r, q, p);
140           }
```

```
141           else {
142             left_right_case(r, q, p);
143           }
144         }
145         else {
146           if (p_on_left) {
147             right_left_case(r, q, p);
148           }
149           else {
150             right_right_case(r, q, p);
151           }
152         }
153       }
154       else {
155         if (p_on_left) {
156           if (q_on_left) {
157             left_left_case(o, p, q);
158           }
159           else {
160             left_right_case(o, p, q);
161           }
162         }
163         else {
164           if (q_on_left) {
165             right_left_case(o, p, q);
166           }
167           else {
168             right_right_case(o, p, q);
169           }
170         }
171       }
172     }
173
174     // swap
175
176     void swap(L const& other) {
177       N* q = other.p;
178       N* x = (*p).parent();
179       N* y = (*q).parent();
180       if ((x == q) or (y == p)) {
181         slide(other);
182         return;
183       }
184       N* x_child[2] = {(*x).left(), (*x).right()};
185       N* p_child[2] = {(*p).left(), (*p).right()};
186       N* y_child[2] = {(*y).left(), (*y).right()};
187       N* q_child[2] = {(*q).left(), (*q).right()};
188       bool p_on_left = ((*x).left() == p);
189       bool q_on_left = ((*y).left() == q);
190       if (q_on_left) {
191         (*y).set_left(p);
192         (*y).set_right(y_child[1]);
```

```
193         (*p).set_left(q_child[0]);
194         (*p).set_right(q_child[1]);
195       }
196       else {
197         (*y).set_left(y_child[0]);
198         (*y).set_right(p);
199         (*p).set_left(q_child[0]);
200         (*p).set_right(q_child[1]);
201       }
202       if (p_on_left) {
203         (*x).set_left(q);
204         (*x).set_right(x_child[1]);
205         (*q).set_left(p_child[0]);
206         (*q).set_right(p_child[1]);
207       }
208       else {
209         (*x).set_left(x_child[0]);
210         (*x).set_right(q);
211         (*q).set_left(p_child[0]);
212         (*q).set_right(p_child[1]);
213       }
214     }
215
216     // operator==
217
218     bool operator==(L const& other) const {
219       return (p == other.p);
220     }
221
222     bool operator==(L_bar const& other) const {
223       return (p == other.p);
224     }
225
226     // operator!=
227
228     bool operator!=(L const& other) const {
229       return not (*this == other);
230     }
231
232     bool operator!=(L_bar const& other) const {
233       return not (*this == other);
234     }
235
236     // print
237
238     void print(std::ostream& stream) const {
239       if (std::is_const<L>::value) {
240         stream << "const navigator: node at " << std::hex << p;
241       }
242       else {
243         stream << "navigator: node at " << std::hex << p;
244       }
```

```
245     }
246
247   private:
248
249     // swap p and q, o: parent of p
250     void left_left_case(N* o, N* p, N* q) {
251       N* r = (*o).right();
252       N* s = (*p).right();
253       N* t = (*q).left();
254       N* u = (*q).right();
255       (*o).set_left(q);
256       (*o).set_right(r);
257       (*q).set_left(p);
258       (*q).set_right(s);
259       (*p).set_left(t);
260       (*p).set_right(u);
261     }
262
263     void left_right_case(N* o, N* p, N* q) {
264       N* r = (*o).right();
265       N* s = (*p).left();
266       N* t = (*q).left();
267       N* u = (*q).right();
268       (*o).set_left(q);
269       (*o).set_right(r);
270       (*q).set_left(s);
271       (*q).set_right(p);
272       (*p).set_left(t);
273       (*p).set_right(u);
274     }
275
276     void right_left_case(N* o, N* p, N* q) {
277       N* r = (*o).left();
278       N* s = (*p).right();
279       N* t = (*q).left();
280       N* u = (*q).right();
281       (*o).set_left(r);
282       (*o).set_right(q);
283       (*q).set_left(p);
284       (*q).set_right(s);
285       (*p).set_left(t);
286       (*p).set_right(u);
287     }
288
289     void right_right_case(N* o, N* p, N* q) {
290       N* r = (*o).left();
291       N* s = (*p).left();
292       N* t = (*q).left();
293       N* u = (*q).right();
294       (*o).set_left(r);
295       (*o).set_right(q);
296       (*q).set_left(s);
```

```
297       (*q).set_right(p);
298       (*p).set_left(t);
299       (*p).set_right(u);
300     }
301   };
302
303   template <typename T>
304   std::ostream& operator<<(std::ostream& stream, link_navigator<T>
          const& t) {
305     t.print(stream);
306     return stream;
307   }
308 }
309
310 #endif
```

Linked/binary_tree_node.h++

```
1  /*
2    A node used in any binary tree
3
4    Version: Unoptimized
5
6    Author: Jyrki Katajainen © 2015
7  */
8
9  #ifndef __CPHSTL_BINARY_TREE_NODE__
10 #define __CPHSTL_BINARY_TREE_NODE__
11
12 #include <utility> // std::move
13
14 namespace cphstl {
15
16   template <typename V>
17   class binary_tree_node {
18   public:
19
20     using value_type = V;
21
22     explicit binary_tree_node()
23       : element(), first(nullptr), second(nullptr), third(nullptr) {
24     }
25
26     binary_tree_node(V const& v)
27       : element(v), first(nullptr), second(nullptr), third(nullptr) {
28     }
29
30     binary_tree_node(V&& v)
31       : element(std::move(v)), first(nullptr), second(nullptr), third
            (nullptr) {
32     }
```

```
33
34      V const& value() const {
35        return element;
36      }
37
38      binary_tree_node* left() const {
39        return first;
40      }
41
42      binary_tree_node* right() const {
43        return second;
44      }
45
46      binary_tree_node* parent() const {
47        return third;
48      }
49
50      V& value() {
51        return element;
52      }
53
54      void set_left(binary_tree_node* q) {
55        first = q;
56        if (q ≠ nullptr) {
57          (*q).third = this;
58        }
59      }
60
61      void set_right(binary_tree_node* q) {
62        second = q;
63        if (q ≠ nullptr) {
64          (*q).third = this;
65        }
66      }
67
68    private:
69
70      V element;
71      binary_tree_node* first;
72      binary_tree_node* second;
73      binary_tree_node* third;
74    };
75  }
76
77  #endif
```

Linked/compact_binary_tree_node.h++

```
1  /*
2    A node used in compact binary trees
3
```

```
4   Version: Unoptimized
5
6   Author: Jyrki Katajainen © 2015
7 */
8
9 #ifndef __CPHSTL_COMPACT_BINARY_TREE_NODE__
10 #define __CPHSTL_COMPACT_BINARY_TREE_NODE__
11
12 #include <utility> // std::move
13
14 namespace cphstl {
15
16   template <typename V>
17   class compact_binary_tree_node {
18   public:
19
20     using value_type = V;
21
22     explicit compact_binary_tree_node()
23       : element(), first(nullptr), second(nullptr) {
24     }
25
26     compact_binary_tree_node(V const& v)
27       : element(v), first(nullptr), second(nullptr) {
28     }
29
30     compact_binary_tree_node(V&& v)
31       : element(std::move(v)), first(nullptr), second(nullptr) {
32     }
33
34     V const& value() const {
35       return element;
36     }
37
38     compact_binary_tree_node* left() const {
39       return first;
40     }
41
42     compact_binary_tree_node* right() const {
43       bool no_right = (first == nullptr) or (first -> second == this)
             ;
44       if (no_right) {
45         return nullptr;
46       }
47       return first -> second;
48     }
49
50     compact_binary_tree_node* parent() const {
51       bool on_cycle = (second -> second -> first == this);
52       if (on_cycle) {
53         return second -> second;
54       }
```

```
55        return second;
56      }
57
58      V& value() {
59        return element;
60      }
61
62      void set_left(compact_binary_tree_node* q) {
63        first = q;
64        if (q ≠ nullptr) {
65          (*q).second = this;
66        }
67      }
68
69      void set_right(compact_binary_tree_node* q) {
70        if (q == nullptr) {
71          if (first ≠ nullptr) {
72            (*first).second = this;
73          }
74          return;
75        }
76        (*first).second = q;
77        (*q).second = this;
78      }
79
80    private:
81
82      V element;
83      compact_binary_tree_node* first;
84      compact_binary_tree_node* second;
85    };
86  }
87
88  #endif
```

Linked/node_factory.h++

This file is a symbolic link to Referent/node_factory.h++.

Linked/binary_heap.i++

```
1  #include "element.i++" // V
2  #include "comparator.i++" // C
3  #include "experiment.i++"  // STANDARD | COMPACT
4  #include <memory> // std::allocator
5
6  #include "binary_heap.h++"
7  #include "binary_tree_node.h++"
8  #include "compact_binary_tree_node.h++"
9  #include "linked_nearly_complete_binary_tree.h++"
```

```
#include "node_factory.h++"

#ifdef STANDARD

using N = cphstl::binary_tree_node<V>;

#endif

#ifdef COMPACT

using N = cphstl::compact_binary_tree_node<V>;

#endif

using A = std::allocator<V>;
using F = cphstl::node_factory<N, A>;
using T = cphstl::linked_nearly_complete_binary_tree<V, N, F>;
using Q = cphstl::binary_heap<V, T, C>;
```

Std

Std/binary_heap.i++

```
#include "element.i++" // V
#include "comparator.i++" // C
#include "experiment.i++" // S
#include <queue> // std::priority_queue

using Q = std::priority_queue<V, S, C>;
```

Williams

This directory is intensionally empty.

Jensen

Jensen/binary_heap.h++

```
/*
  Authors: Claus Jensen, Jyrki Katajainen © 2006, 2015

  An implementation of binary heaps is provided which stores pointers
  to the elements instead the elements themselves.
*/

#include <algorithm> // std::make_heap, std::push_heap, std::
    pop_heap
```

```
#include <cstddef> // std::size_t
#include <functional> // std::less
#include <memory> // std::allocator
#include <vector> // std::vector

template <typename V>
class ElementNode  {
public:

  ElementNode () {
  };

  ElementNode (V const& e) : e(e) {
  };

  V e;
};

template <typename Npointer, typename C>
class less_ref  {
protected:
  C _less;

public:

  explicit less_ref() {
  }

  explicit less_ref(C const& less) : _less(less) {
  }

  bool operator()(Npointer const& x, Npointer const& y) const {
    return _less(x->e, y->e);
  }

  less_ref& operator=(C const& less) const {
    _less = less;
    return *this;
  }
};

template <
  typename V,
  typename S = std::vector<V, std::allocator<V> >,
  typename C = std::less<typename S::value_type>
>
class binary_heap {
private:

  typedef ElementNode<V> N;

  template <typename T, typename U, typename Some>
```

```
61    struct rebind;
62
63    template <typename T, typename U, template <typename...> class Name
          , typename... Args>
64    struct rebind<T, U, Name<Args...>> {
65      typedef Name<T, U> other;
66    };
67
68    using A = typename S::allocator_type;
69    using A_prime = typename A::template rebind<N>::other;
70    using A_prime_prime = typename A::template rebind<N*>::other;
71    using S_prime_prime = typename rebind<N*, A_prime_prime, S>::other;
72
73    S_prime_prime _P;
74    less_ref<N*, C> _comp;
75
76  public:
77
78    typedef typename S::value_type value_type;
79    typedef typename S::size_type size_type;
80    typedef S container_type;
81
82    explicit binary_heap(C const& x = C())
83      : _P(), _comp(x) {
84    }
85
86    template <typename X>
87    binary_heap(C const& x, X const& Y)
88      : _P(), _comp(x) {
89      for (typename X::const_iterator i = Y.begin(); i != Y.end(); ++i
            ) {
90        N* en = new ElementNode<V>;
91        en->e = *i;
92        _P.push_back(en);
93      }
94      std::make_heap(_P.begin(), _P.end(), _comp);
95    }
96
97    template <typename InputIterator>
98    binary_heap(InputIterator first, InputIterator last, C const& x = C
          (),
99       S const& Y = S());
100
101   ~binary_heap();
102
103   bool empty() const;
104   size_type size() const;
105   V const& top() const;
106   void push(V const& x);
107   void pop();
108 };
109
```

```
#include "binary_heap.c++"
```

Jensen/binary_heap.c++

```
/*
  Author: Claus Jensen @ 2006
*/

template <typename V, typename S, typename C>
template <typename InputIterator>
binary_heap<V, S, C>::binary_heap(InputIterator first, InputIterator
    last,
                    const C& x,  const S& Y) {
  _comp = less_ref<N*, C>(x);
  for (typename S::const_iterator i = Y.begin(); i != Y.end(); i++)
      {
    N* en = new ElementNode<V>;
    en->e = *i;
    _P.push_back(en);
  }
  for (InputIterator i = first; i != last; i++) {
    N* en = new ElementNode<V>;
    en->e = *i;
    _P.push_back(en);
  }
  std::make_heap(_P.begin(), _P.end(), _comp);
}

template <typename V, typename S, typename C>
binary_heap<V, S, C>::~binary_heap() {
  for (typename S_prime_prime::iterator i = _P.begin(); i != _P.end()
      ; i++) {
    N* b = *i;
    delete b;
  }
}

template <typename V, typename S, typename C>
bool binary_heap<V, S, C>::empty() const {
  return(_P.empty());
}

template <typename V, typename S, typename C>
typename binary_heap<V, S, C>::size_type
binary_heap<V, S, C>::size() const {
  return(_P.size());
}

template <typename V, typename S, typename C>
V const& binary_heap<V, S, C>::top() const {
  return(_P[0]->e);
```

```
45 }
46
47 template <typename V, typename S, typename C>
48 void binary_heap<V, S, C>::push(V const& x) {
49   N* en = new ElementNode<V>;
50   en->e = x;
51   _P.push_back(en);
52   std::push_heap(_P.begin(), _P.end(), _comp);
53 }
54
55 template <typename V, typename S, typename C>
56 void binary_heap<V, S, C>::pop() {
57   N* i = _P[0];
58   std::pop_heap(_P.begin(), _P.end(), _comp);
59   _P.pop_back();
60   delete i;
61 }
```

Jensen/binary_heap.i++

```
1 #include "element.i++" // V
2 #include "comparator.i++" // C
3 #include "experiment.i++"  // S
4 #include "binary_heap.h++" // Jensen's referent binary_heap
5
6 using Q = binary_heap<V, S, C>;
```

Edelkamp-Katajainen

Edelkamp-Katajainen/binary_heap.i++

```
1 #include "element.i++" // V
2 #include "comparator.i++" // C
3 #include "experiment.i++"  // S
4
5 #include "element_encapsulator.h++"
6 #include <memory>
7 #include "single_heap_framework.h++"
8 #include "meldable_priority_queue.h++"
9 #include "top_down_binary_heap_heapifier.h++"
10 #include "vector_encapsulator.h++"
11 #include "unidirectional_monolith_iterator.h++"
12
13 template <typename T, typename U, typename Some>
14 struct rebind;
15
16 template <typename T, typename U, template <typename...> class Name,
       typename... Args>
17 struct rebind<T, U, Name<Args...>> {
```

```
18    using other = Name<T, U>;
19  };
20
21  using A = std::allocator<V>;
22  using N = cphstl::element_encapsulator<V, A>;
23  using H = cphstl::top_down_binary_heap_heapifier;
24
25  using A_prime = typename A::template rebind<N*>::other;
26  using S_prime = typename rebind<N*, A_prime, S>::other;
27  using B = cphstl::vector_encapsulator<S_prime, A_prime>;
28
29  using R = cphstl::single_heap_framework<V, C, A, N, H, S_prime, B>;
30
31  using I = cphstl::unidirectional_monolith_iterator<R>;
32  using J = cphstl::unidirectional_monolith_iterator<R, true>;
33
34  using Q = cphstl::meldable_priority_queue<V, C, A, N, R, I, J>;
```

Goodrich-et-al

This directory is intensionally empty.

Tuned-implicit package

Tuned-implicit/binary_heap.h++

```
1  /*
2    Binary–heap class template for all–in–one implementation framework
3
4    Version: Tuned; for the eyes of the library implementers only
5
6    Author: Jyrki Katajainen © 2014, 2015
7  */
8
9  #include <cstddef> // std::size_t
10 #include <iostream> // std streams
11 #include <utility> // std::move
12
13 namespace cphstl {
14
15   template <typename V, typename T, typename C>
16   class binary_heap {
17   public:
18
19     // types
20
21     using value_type = V;
22     using tree_type = T;
```

```
23      using comparator_type = C;
24      using size_type = std::size_t;
25
26    private:
27
28      using M = typename T::const_navigator;
29      using N = typename T::navigator;
30
31      // constants
32
33      M none;
34      enum class direction : size_type {left = 0, right = 1, parent =
            2};
35
36      // variables
37
38      T tree;
39      C less;
40
41    public:
42
43      // structors
44
45      explicit binary_heap(C const& c = C())
46        : tree(), less(c) {
47      }
48
49      template <typename S>
50      binary_heap(C const& c, S const& s)
51        : tree(s), less(c) {
52        if (tree.size() != 0) {
53          build(tree.root());
54        }
55      }
56
57  /*
58    binary_heap(C const& c, typename T::sequence_type&& s)
59        : tree(std::move(s)), less(c) {
60        if (tree.size() != 0) {
61          build(tree.root());
62        }
63      }
64  */
65      ~binary_heap() {
66      }
67
68      binary_heap(binary_heap const& other)
69        : tree(other.tree), less(other.less) {
70      }
71
72      binary_heap(binary_heap&& other)
73        : tree(std::move(other.tree)), less(other.less) {
```

```
74      }
75
76      binary_heap& operator=(binary_heap const& other) {
77        less = other.less;
78        binary_heap tmp(other);
79        clear();
80        tree.swap(tmp.tree);
81        return *this;
82      }
83
84      binary_heap& operator=(binary_heap&& other) {
85        less = other.less;
86        binary_heap tmp(std::move(other));
87        clear();
88        tree.swap(tmp.tree);
89        return *this;
90      }
91
92      // accessors
93
94      size_type size() const {
95        return tree.size();
96      }
97
98      V const& top() const {
99        return *tree.root();
100     }
101
102     // modifiers
103
104     void push(V const& in) {
105       N hole = tree.expand();
106       siftup(hole, in);
107       *hole = in;
108     }
109
110     void push(V&& in) {
111       N hole = tree.expand();
112       siftup(hole, in);
113       *hole = std::move(in);
114     }
115
116     void pop() {
117       if (tree.size() < 3) {
118         if (tree.size() == 2) {
119           N old_root = tree.root();
120           N leaf = tree.detach();
121           old_root.replace(leaf);
122           tree.remove(old_root);
123         }
124         else {
125           tree.contract();
```

```
126         }
127       }
128       else {
129         N hole = tree.root();
130         N leaf = tree.last();
131         if (tree.size() bitand 1) {
132           siftdown(hole, *leaf);
133           leaf = tree.detach();
134         }
135         else {
136           leaf = tree.detach();
137           siftdown(hole, *leaf);
138         }
139         hole.replace(leaf);
140         tree.remove(hole);
141       }
142     }
143
144     void pop(V& out) {
145       std::cout << "pop: " << *tree.root() << std::endl;
146       out = std::move(*tree.root());
147       pop();
148     }
149
150     void clear() {
151       while (tree.size() ≠ 0) {
152         N leaf = tree.detach();
153         tree.remove(leaf);
154       }
155     }
156
157   private:
158
159     void siftup(N& hole, V const& in) {
160       while (not hole.is_root()) {
161         N p = hole.parent();
162         if (not less(*p, in)) {
163           break;
164         }
165         hole.slide(p);
166       }
167     }
168
169     N siftdown(N filled) {
170       N p = filled.left();
171       N q = filled.right();
172       if (less(*p, *q)) {
173         if (less(*filled, *q)) {
174           V substitute = std::move(*filled);
175           filled.slide(q);
176           N subtree_root = filled.parent();
177           siftdown(filled, substitute);
```

```
178           *filled = std::move(substitute);
179           return subtree_root;
180         }
181         return filled;
182       }
183       if (less(*filled, *p)) {
184         V substitute = std::move(*filled);
185         filled.slide(p);
186         N subtree_root = filled.parent();
187         siftdown(filled, substitute);
188         *filled = std::move(substitute);
189         return subtree_root;
190       }
191       return filled;
192     }
193
194     void siftdown(N& hole, V const& in) {
195       while (not hole.is_leaf()) {
196         N p = hole.left();
197         N q = hole.right();
198         if (less(*p, *q)) {
199           if (not less(in, *q)) {
200             break;
201           }
202           hole.slide(q);
203         }
204         else {
205           if (not less(in, *p)) {
206             break;
207           }
208           hole.slide(p);
209         }
210       }
211     }
212
213     void build(N current) {
214       size_type const n = tree.size();
215       if (n ≤ 2) {
216         if (n == 2) {
217           N hole = tree.last();
218           V tmp = std::move(*hole);
219           siftup(hole, tmp);
220           *hole = std::move(tmp);
221         }
222         return;
223       }
224       size_type const m = (n bitand 1) ? n : n − 1;
225       N leaf = tree.last();
226       if (m ≠ n) {
227         leaf = tree.detach();
228       }
229       direction from = direction::parent;
```

```
230       size_type bit_stack = 0;
231       for (size_type i = 0; i != 2 * m - 2; ++i) { // repeat 2 m - 2
              times
232         if (from == direction::left) {
233           current = current.right();
234           bit_stack = bit_stack << 1;
235           bit_stack = bit_stack bitor size_type(direction::right); //
                push
236           from = direction::parent;
237         }
238         else if (from == direction::right) {
239           current = siftdown(current);
240           current = current.parent();
241           from = direction(bit_stack bitand 1); // top
242           bit_stack >>= 1; // pop
243         }
244         else if (current.is_leaf()) {
245           current = current.parent();
246           from = direction(bit_stack bitand 1); // top
247           bit_stack >>= 1; // pop
248         }
249         else {
250           current = current.left();
251           from = direction::parent;
252           bit_stack = bit_stack << 1;
253           bit_stack = bit_stack bitor size_type(direction::left); //
                push
254         }
255       }
256       siftdown(current);
257       if (m != n) {
258         tree.attach(leaf);
259         N hole = tree.last();
260         V tmp = std::move(*hole);
261         siftup(hole, tmp);
262         *hole = std::move(tmp);
263       }
264     }
265   };
266 }
```

Tuned-implicit/implicit_nearly_complete_binary_tree.h++

```
1 /*
2   An implicit nearly-complete binary tree needed in the
3   implementation framework for binary heaps
4
5   Version: Tuned
6
7   Author: Jyrki Katajainen @ 2012, 2013, 2015
8 */
```

```
9
10 #ifndef __CPHSTL_IMPLICIT_NEARLY_COMPLETE_BINARY_TREE__
11 #define __CPHSTL_IMPLICIT_NEARLY_COMPLETE_BINARY_TREE__
12
13 #include <cstddef> // std::size_t
14 #include <memory> // std::allocator
15 #include "rank_navigator.h++"
16 #include <vector> // std::vector
17 #include <utility> // std::move
18
19 namespace cphstl {
20
21   template <typename V, typename S = std::vector<V, std::allocator<V
       >>>
22   class implicit_nearly_complete_binary_tree {
23   public:
24
25     using value_type = V;
26     using sequence_type = S;
27     using allocator_type = typename S::allocator_type;
28     using reference = V&;
29     using const_reference = V const&;
30     using size_type = std::size_t;
31     using self_type = cphstl::implicit_nearly_complete_binary_tree<V,
           S>;
32     using navigator = cphstl::rank_navigator<self_type>;
33     using const_navigator = cphstl::rank_navigator<self_type const>;
34
35   private:
36
37     friend navigator;
38     friend const_navigator;
39
40     S sequence;
41     bool one_outside;
42
43     const_reference operator[](size_type i) const {
44       return sequence[i];
45     }
46
47     reference operator[](size_type i) {
48       return sequence[i];
49     }
50
51   public:
52
53     explicit implicit_nearly_complete_binary_tree()
54       : sequence(), one_outside(false) {
55     }
56
57     implicit_nearly_complete_binary_tree(S const& s)
58       : sequence(s), one_outside(false) {
```

```
59      }
60
61      implicit_nearly_complete_binary_tree(S&& s)
62        : sequence(std::move(s)), one_outside(false) {
63      }
64
65      template <typename X>
66      implicit_nearly_complete_binary_tree(X const& s)
67        : sequence(s.get_allocator()), one_outside(false) {
68        sequence.reserve(s.size());
69        try {
70          for (V x : s) {
71            sequence.push_back(x);
72          }
73        }
74        catch (...) {
75          sequence.clear();
76          throw;
77        }
78      }
79
80      implicit_nearly_complete_binary_tree(self_type const& other)
81        : sequence(other.sequence), one_outside(other.one_outside) {
82      }
83
84      implicit_nearly_complete_binary_tree(self_type&& other)
85        : sequence(std::move(other.sequence)), one_outside(other.
              one_outside) {
86      }
87
88      ~implicit_nearly_complete_binary_tree() {
89        clear();
90      }
91
92      self_type& operator=(self_type const& other) {
93        self_type tmp(other);
94        clear();
95        swap(tmp);
96        return *this;
97      }
98
99      self_type& operator=(self_type&& other) {
100       self_type tmp(std::move(other));
101       clear();
102       swap(tmp);
103       return *this;
104     }
105
106     allocator_type get_allocator() const {
107       return sequence.get_allocator();
108     }
109
```

```
110     size_type size() const {
111       return sequence.size() − one_outside;
112     }
113
114     const_navigator root() const {
115       return const_navigator(this, 0);
116     }
117
118     navigator root() {
119       return navigator(this, 0);
120     }
121
122     const_navigator last() const {
123       return const_navigator(this, size() − 1);
124     }
125
126     navigator last() {
127       return navigator(this, size() − 1);
128     }
129
130     navigator expand() {
131       sequence.push_back(std::move(V()));
132       return last();
133     }
134
135     void attach(navigator /* not used */) {
136       one_outside = false;
137     }
138
139     navigator detach() {
140       navigator past = last();
141       one_outside = true;
142       return past;
143     }
144
145     void remove(navigator /* not used */) {
146       contract();
147     }
148
149     void contract() {
150       one_outside = false;
151       sequence.pop_back();
152     }
153
154     void swap(self_type& other) {
155       sequence.swap(other.sequence);
156       std::swap(one_outside, other.one_outside);
157     }
158
159     void clear() {
160       sequence.clear();
161       one_outside = false;
```

```
162     }
163   };
164 }
165
166 #endif
```

Tuned-implicit/rank_navigator.h++

```
1  /*
2    A rank navigator encapsulates a position of a value by storing a
3    (pointer, rank) pair; the pointer refers to the container—the
4    owner—that contains the value referred to and the rank is the
5    index of that value within the container.
6
7    Version: Tuned
8
9    Authors: Jyrki Katajainen, Andreas Milton Maniotis, Bo Simonsen
10   © 2008, 2012, 2013, 2015
11 */
12
13 #ifndef __CPHSTL_RANK_NAVIGATOR__
14 #define __CPHSTL_RANK_NAVIGATOR__
15
16 #include <type_traits> // std::conditional, std::is_const, std::
       remove_const
17 #include <utility> // std::move, std::swap
18
19 namespace cphstl {
20
21   template <typename T>
22   class rank_navigator {
23
24   public:
25
26     // type aliases
27
28     using owner_type = T;
29
30   private:
31
32     using T_bar = typename std::conditional<std::is_const<T>::value,
           typename std::remove_const<T>::type, T const>::type;
33     using R = rank_navigator<T>;
34     using R_bar = rank_navigator<T_bar>;
35     using size_type = typename T::size_type;
36     using V = typename T::value_type;
37     using V_check = typename std::conditional<std::is_const<T>::value
           , V const, V>::type;
38     using pointer = V_check*;
39     using reference = V_check&;
40
```

```
41      // friends
42
43      friend T;
44      friend R_bar;
45
46      // constants
47
48      static size_type const outside = -1;
49
50      // variables
51
52      T* owner_pointer;
53      size_type rank;
54
55      // parameterized constructor
56
57      rank_navigator(owner_type* p, size_type offset)
58        : owner_pointer(p), rank(offset) {
59      }
60
61    public:
62
63      // default constructor
64
65      rank_navigator()
66        : owner_pointer(nullptr), rank(outside) {
67      }
68
69      // copy constructors
70
71      // Generated by the compiler if needed
72      // rank_navigator(const rank_navigator&) = default;
73
74      rank_navigator(rank_navigator<typename std::remove_const<T>::type
            > const& x)
75        : owner_pointer(x.owner_pointer), rank(x.rank) {
76      }
77
78      // assignments
79
80      // Generated by the compiler if needed
81      // rank_navigator& operator=(rank_navigator const&) = default;
82
83      rank_navigator& operator=(rank_navigator<typename std::
            remove_const<T>::type> const& x) {
84        owner_pointer = x.owner_pointer;
85        rank = x.rank;
86        return *this;
87      }
88
89      // destructor
90
```

```
91      ~rank_navigator() {
92      };
93
94      // operator*
95
96      reference operator*() const {
97        return (*owner_pointer)[rank];
98      }
99
100     // operator->
101
102     pointer operator->() const {
103       return &(*owner_pointer)[rank];
104     }
105
106     // left
107
108     R left() const {
109       return R(owner_pointer, 2 * rank + 1);
110     }
111
112     // right
113
114     R right() const {
115       return R(owner_pointer, 2 * rank + 2);
116     }
117
118     // parent
119
120     R parent() const {
121       return R(owner_pointer, (rank - 1) / 2);
122     }
123
124     // slide
125
126     void slide(R const& neighbour) {
127       replace(neighbour);
128     }
129
130     // replace
131
132     void replace(R const& replacement) {
133       **this = std::move(*replacement);
134       rank = replacement.rank;
135     }
136
137     // operator==
138
139     bool operator==(R const& other) const {
140       return (rank == other.rank) and (owner_pointer == other.
              owner_pointer);
141     }
```

```
142
143     bool operator==(R_bar const& other) const {
144       return (rank == other.rank) and (owner_pointer == other.
              owner_pointer);
145     }
146
147     // operator≠
148
149     bool operator≠(R const& other) const {
150       return not (*this == other);
151     }
152
153     bool operator≠(R_bar const& other) const {
154       return not (*this == other);
155     }
156
157     // is_root
158
159     bool is_root() const {
160       return (rank == 0);
161     }
162
163     // is_leaf
164
165     bool is_leaf() const {
166       return 2 * rank + 1 ≥ (*owner_pointer).size();
167     }
168
169     // has_right
170
171     bool has_right() const {
172       return (2 * rank + 2 < (*owner_pointer).size());
173     }
174   };
175 }
176
177 #endif
```

Tuned-implicit/binary_heap.i++

This file is a symbolic link to Implicit/binary_heap.i++.

Tuned-referent package

Tuned-referent/binary_heap.h++

This file is a symbolic link to Tuned-implicit/binary_heap.h++.

Tuned-referent/referent_nearly_complete_binary_tree.h++

```
/*
  A referent nearly-complete binary tree needed in the implementation
  framework for binary heaps

  Version: Tuned

  Author: Jyrki Katajainen @ 2012, 2013, 2015
*/

#ifndef __CPHSTL_REFERENT_NEARLY_COMPLETE_BINARY_TREE__
#define __CPHSTL_REFERENT_NEARLY_COMPLETE_BINARY_TREE__

#include <cstddef> // std::size_t
#include <memory> // std::allocator
#include "node_factory.h++"
#include "rank_navigator.h++"
#include "value_encapsulator.h++"
#include <vector> // std::vector
#include <utility> // std::move

namespace cphstl {

  template <
    typename V,
    typename S = std::vector<V, std::allocator<V>>,
    typename N = cphstl::value_encapsulator<V>,
    typename F = cphstl::node_factory<N, typename S::allocator_type>
  >
  class referent_nearly_complete_binary_tree {
  public:

    using value_type = V;
    using sequence_type = S;
    using node_type = N;
    using factory_type = F;
    using allocator_type = typename S::allocator_type;
    using reference = V&;
    using const_reference = V const&;
    using size_type = std::size_t;
    using self_type = cphstl::referent_nearly_complete_binary_tree<V,
        S, N, F>;
    using navigator = cphstl::rank_navigator<self_type>;
    using const_navigator = cphstl::rank_navigator<self_type const>;

  private:

    friend navigator;
    friend const_navigator;

```

```
49    template <typename T, typename U, typename Some>
50    struct rebind;
51
52    template <typename T, typename U, template <typename...> class
          Name, typename... Args>
53    struct rebind<T, U, Name<Args...>> {
54      typedef Name<T, U> other;
55    };
56
57    using A = typename S::allocator_type;
58    using A_prime = typename A::template rebind<N>::other;
59    using A_prime_prime = typename A::template rebind<N*>::other;
60    using S_prime_prime = typename rebind<N*, A_prime_prime, S>::
          other;
61    using F_prime = typename rebind<N, A_prime, F>::other;
62
63    S_prime_prime sequence;
64    F_prime factory;
65    bool one_outside;
66
67    const_reference operator[](size_type i) const {
68      return (*sequence[i]).value();
69    }
70
71    reference operator[](size_type i) {
72      return (*sequence[i]).value();
73    }
74
75  public:
76
77    explicit referent_nearly_complete_binary_tree(F const& f = F())
78      : sequence(f.get_allocator()), factory(f), one_outside(false) {
79    }
80
81    template <typename X>
82    referent_nearly_complete_binary_tree(X const& s, F const& f = F()
          )
83      : sequence(s.get_allocator()), factory(f), one_outside(false) {
84      sequence.reserve(s.size());
85      try {
86        for (V x : s) {
87          sequence.push_back(factory.create(x));
88        }
89      }
90      catch (...) {
91        clear();
92        throw;
93      }
94    }
95
96    referent_nearly_complete_binary_tree(self_type&& other)
```

```
97      : sequence(other.get_allocator()), factory(other.factory),
            one_outside(false) {
98      swap(other);
99    }
100
101   ~referent_nearly_complete_binary_tree() {
102     clear();
103   }
104
105   self_type& operator=(self_type const& other) {
106     self_type tmp(other);
107     clear();
108     swap(tmp);
109     return *this;
110   }
111
112   self_type& operator=(self_type&& other) {
113     self_type tmp(std::move(other));
114     clear();
115     swap(tmp);
116     return *this;
117   }
118
119   allocator_type get_allocator() const {
120     return sequence.get_allocator();
121   }
122
123   size_type size() const {
124     return sequence.size() - one_outside;
125   }
126
127   const_navigator root() const {
128     return const_navigator(this, 0);
129   }
130
131   navigator root() {
132     return navigator(this, 0);
133   }
134
135   const_navigator last() const {
136     return const_navigator(this, size() - 1);
137   }
138
139   navigator last() {
140     return navigator(this, size() - 1);
141   }
142
143   navigator expand() {
144     N* p = factory.create();
145     sequence.push_back(p);
146     return last();
147   }
```

```
148
149     void attach(navigator /* not used */) {
150       one_outside = false;
151     }
152
153     navigator detach() {
154       navigator past = last();
155       one_outside = true;
156       return past;
157     }
158
159     void remove(navigator /* not used */) {
160       contract();
161     }
162
163     void contract() {
164       one_outside = false;
165       size_type ℓ = size() - 1;
166       N* p = sequence[ℓ];
167       factory.destroy(p);
168       sequence.pop_back();
169     }
170
171     void swap(self_type& other) {
172       sequence.swap(other.sequence);
173       factory.swap(other.factory);
174       std::swap(one_outside, other.one_outside);
175     }
176
177     void clear() {
178       for (N* p : sequence) {
179         factory.destroy(p);
180       }
181       sequence.clear();
182       one_outside = false;
183     }
184   };
185 }
186
187 #endif
```

Tuned-referent/rank_navigator.h++

This file is a symbolic link to Tuned-implicit/rank_navigator.h++.

Tuned-referent/value_encapsulator.h++

This file is a symbolic link to Referent/value_encapsulator.h++.

Tuned-referent/node_factory.h++

This file is a symbolic link to Referent/node_factory.h++.

Tuned-referent/binary_heap.i++

This file is a symbolic link to Referent/binary_heap.i++.

Tuned-inverse package

Tuned-inverse/binary_heap.h++

This file is a symbolic link to Tuned-implicit/binary_heap.h++.

Tuned-inverse/inverse_nearly_complete_binary_tree.h++

```
/*
  An inverse nearly–complete binary tree needed in the implementation
  framework for binary heaps

  Version: Tuned

  Author: Jyrki Katajainen @ 2012, 2013, 2015
*/

#ifndef __CPHSTL_INVERSE_NEARLY_COMPLETE_BINARY_TREE__
#define __CPHSTL_INVERSE_NEARLY_COMPLETE_BINARY_TREE__

#include <cstddef> // std::size_t
#include "inverse_link_navigator.h++"
#include "inverse_tree_node.h++"
#include <memory> // std::allocator
#include "node_factory.h++"
#include <vector> // std::vector
#include <utility> // std::move

namespace cphstl {

  template <
    typename V,
    typename S = std::vector<V, std::allocator<V>>,
    typename N = cphstl::inverse_tree_node<V>,
    typename F = cphstl::node_factory<N, typename S::allocator_type>
  >
  class inverse_nearly_complete_binary_tree {
  public:

    using value_type = V;
```

```
33    using sequence_type = S;
34    using node_type = N;
35    using factory_type = F;
36    using allocator_type = typename S::allocator_type;
37    using reference = V&;
38    using const_reference = V const&;
39    using size_type = std::size_t;
40    using self_type = cphstl::inverse_nearly_complete_binary_tree<V,
          S, N, F>;
41    using navigator = cphstl::inverse_link_navigator<self_type>;
42    using const_navigator = cphstl::inverse_link_navigator<self_type
          const>;
43
44  private:
45
46    friend navigator;
47    friend const_navigator;
48
49    template <typename T, typename U, typename Some>
50    struct rebind;
51
52    template <typename T, typename U, template <typename...> class
          Name, typename... Args>
53    struct rebind<T, U, Name<Args...>> {
54      typedef Name<T, U> other;
55    };
56
57    using A = typename S::allocator_type;
58    using A_prime = typename A::template rebind<N>::other;
59    using A_prime_prime = typename A::template rebind<N*>::other;
60    using S_prime_prime = typename rebind<N*, A_prime_prime, S>::
          other;
61    using F_prime = typename rebind<N, A_prime, F>::other;
62
63    S_prime_prime sequence;
64    F_prime factory;
65
66    N* const& operator[](size_type i) const {
67      return sequence[i];
68    }
69
70    N*& operator[](size_type i) {
71      return sequence[i];
72    }
73
74  public:
75
76    explicit inverse_nearly_complete_binary_tree(F const& f = F())
77      : sequence(f.get_allocator()), factory(f) {
78    }
79
80    template <typename X>
```

```
81    inverse_nearly_complete_binary_tree(X const& s, F const& f = F())
82      : sequence(s.get_allocator()), factory(f) {
83      sequence.reserve(s.size());
84      size_type i = 0;
85      try {
86        for (V x : s) {
87          sequence.push_back(factory.create(x, i));
88          i += 1;
89        }
90      }
91      catch (...) {
92        while (i ≠ 0) {
93          --i;
94          factory.destroy(sequence[i]);
95        }
96        sequence.clear();
97        throw;
98      }
99    }
100
101   inverse_nearly_complete_binary_tree(self_type&& other)
102     : sequence(other.get_allocator()), factory(other.factory) {
103     swap(other);
104   }
105
106   ~inverse_nearly_complete_binary_tree() {
107     clear();
108   }
109
110   self_type& operator=(self_type const& other) {
111     self_type tmp(other);
112     clear();
113     swap(tmp);
114     return *this;
115   }
116
117   self_type& operator=(self_type&& other) {
118     self_type tmp(std::move(other));
119     clear();
120     swap(tmp);
121     return *this;
122   }
123
124   allocator_type get_allocator() const {
125     return sequence.get_allocator();
126   }
127
128   size_type size() const {
129     return sequence.size();
130   }
131
132   const_navigator root() const {
```

```
133       return const_navigator(this, sequence[0]);
134     }
135
136     navigator root() {
137       return navigator(this, sequence[0]);
138     }
139
140     const_navigator last() const {
141       return const_navigator(this, sequence[size() - 1]);
142     }
143
144     navigator last() {
145       return navigator(this, sequence[size() - 1]);
146     }
147
148     navigator expand() {
149       size_type past = size();
150       N* p = factory.create(std::move(V()), past);
151       sequence.push_back(p);
152       return navigator(this, p);
153     }
154
155     void attach(navigator outsider) {
156       size_type past = size();
157       N* p = outsider.p;
158       (*p).rank() = past;
159       sequence.push_back(p);
160     }
161
162     navigator detach() {
163       size_type ℓ = size() - 1;
164       N* p = sequence[ℓ];
165       sequence.pop_back();
166       return navigator(nullptr, p);
167     }
168
169     void remove(navigator outsider) {
170       factory.destroy(outsider.p);
171     }
172
173     void contract() {
174       size_type ℓ = size() - 1;
175       N* p = sequence[ℓ];
176       factory.destroy(p);
177       sequence.pop_back();
178     }
179
180     void swap(self_type& other) {
181       sequence.swap(other.sequence);
182       factory.swap(other.factory);
183     }
184
```

```
185     void clear() {
186       for (N* p : sequence) {
187         factory.destroy(p);
188       }
189       sequence.clear();
190     }
191   };
192 }
193
194 #endif
```

Tuned-inverse/inverse_link_navigator.h++

```
1  /*
2    An inverse link navigator encapsulates a position of a value by
3    storing a pair of pointers, one to its owner and another to the
4    node where the value is stored
5
6    Version: Tuned
7
8    Author: Jyrki Katajainen © 2015
9  */
10
11 #ifndef __CPHSTL_INVERSE_LINK_NAVIGATOR__
12 #define __CPHSTL_INVERSE_LINK_NAVIGATOR__
13
14 #include <limits> // std::numeric_limits
15 #include <type_traits> // std::conditional, std::is_const, std::
       remove_const
16
17 namespace cphstl {
18
19   template <typename T>
20   class inverse_link_navigator {
21
22   public:
23
24     // type aliases
25
26     using owner_type = T;
27
28   private:
29
30     using T_bar = typename std::conditional<std::is_const<T>::value,
           typename std::remove_const<T>::type, T const>::type;
31     using L = inverse_link_navigator<T>;
32     using L_bar = inverse_link_navigator<T_bar>;
33     using size_type = typename T::size_type;
34     using V = typename T::value_type;
35     using N = typename T::node_type;
```

```
36      using V_check = typename std::conditional<std::is_const<T>::value
            , V const, V>::type;
37      using pointer = V_check*;
38      using reference = V_check&;
39
40      // friends
41
42      friend T;
43      friend L_bar;
44
45      // variables
46
47      T* owner_pointer;
48      N* p;
49
50      // parameterized constructor
51
52      inverse_link_navigator(T* p, N* x)
53        : owner_pointer(p), p(x) {
54      }
55
56    public:
57
58      // default constructor
59
60      inverse_link_navigator()
61        : owner_pointer(nullptr), p(nullptr) {
62      }
63
64      // copy constructors
65
66      // Generated by the compiler if needed
67      // inverse_link_navigator(const inverse_link_navigator&) =
            default;
68
69      inverse_link_navigator(inverse_link_navigator<typename std::
            remove_const<T>::type> const& x)
70        : owner_pointer(x.owner_pointer), p(x.p) {
71      }
72
73      // assignments
74
75      // Generated by the compiler if needed
76      // inverse_link_navigator& operator=(inverse_link_navigator const
            &) = default;
77
78      inverse_link_navigator& operator=(inverse_link_navigator<typename
            std::remove_const<T>::type> const& x) {
79        owner_pointer = x.owner_pointer;
80        p = x.p;
81        return *this;
82      }
```

```
83
84      // destructor
85
86      ~inverse_link_navigator() {
87      };
88
89      // operator*
90
91      reference operator*() const {
92        return (*p).value();
93      }
94
95      // operator->
96
97      pointer operator->() const {
98        return &(*p).value();
99      }
100
101     // left
102
103     L left() const {
104       size_type rank = 2 * (*p).rank() + 1;
105       return L(owner_pointer, (*owner_pointer)[rank]);
106     }
107
108     // right
109
110     L right() const {
111       size_type rank = 2 * (*p).rank() + 2;
112       return L(owner_pointer, (*owner_pointer)[rank]);
113     }
114
115     // parent
116
117     L parent() const {
118       size_type rank = ((*p).rank() - 1) / 2;
119       return L(owner_pointer, (*owner_pointer)[rank]);
120     }
121
122     // slide
123
124     void slide(L const& neighbour) {
125       size_type i = (*p).rank();
126       size_type j = (*neighbour.p).rank();
127       (*p).rank() = j;
128       (*neighbour.p).rank() = i;
129       (*owner_pointer)[i] = neighbour.p;
130       (*owner_pointer)[j] = p;
131     }
132
133     // replace
134
```

```
135     void replace(L& replacement) {
136       replacement.owner_pointer = owner_pointer;
137       size_type i = (*p).rank();
138       (*replacement.p).rank() = i;
139       (*owner_pointer)[i] = replacement.p;
140       owner_pointer = nullptr;
141     }
142
143     // operator==
144
145     bool operator==(L const& other) const {
146       return p == other.p;
147     }
148
149     bool operator==(L_bar const& other) const {
150       return p == other.p;
151     }
152
153     // operator≠
154
155     bool operator≠(L const& other) const {
156       return not (*this == other);
157     }
158
159     bool operator≠(L_bar const& other) const {
160       return not (*this == other);
161     }
162
163     // is_root
164
165     bool is_root() const {
166       return (*p).rank() == 0;
167     }
168
169     // is_leaf
170
171     bool is_leaf() const {
172       return 2 * (*p).rank() + 1 ≥ (*owner_pointer).size();
173     }
174
175     // has_right
176
177     bool has_right() const {
178       return 2 * (*p).rank() + 2 < (*owner_pointer).size();
179     }
180   };
181 }
182
183 #endif
```

Tuned-inverse/inverse_tree_node.h++

This file is a symbolic link to Inverse/inverse_tree_node.h++.

Tuned-inverse/node_factory.h++

This file is a symbolic link to Referent/node_factory.h++.

Tuned-inverse/binary_heap.i++

This file is a symbolic link to Inverse/binary_heap.i++.

Tuned-linked package

Tuned-linked/binary_heap.h++

This file is a symbolic link to Tuned-implicit/binary_heap.h++.

Tuned-linked/linked_nearly_complete_binary_tree.h++

```
/*
  A linked nearly-complete binary tree needed in the implementation
  framework for binary heaps

  Version: Tuned

  Author: Jyrki Katajainen @ 2012, 2013, 2015
*/

#ifndef __CPHSTL_LINKED_NEARLY_COMPLETE_BINARY_TREE__
#define __CPHSTL_LINKED_NEARLY_COMPLETE_BINARY_TREE__

#include "binary_tree_node.h++"
#include <cmath> // ilogb
#include <cstddef> // std::size_t
#include "link_navigator.h++"
#include <memory> // std::allocator
#include "node_factory.h++"
#include <utility> // std::swap, std::move

namespace cphstl {

  template <
    typename V,
    typename N = cphstl::binary_tree_node<V>,
    typename F = cphstl::node_factory<N, std::allocator<N>>
  >
```

```
28  class linked_nearly_complete_binary_tree {
29  public:
30
31    using value_type = V;
32    using node_type = N;
33    using factory_type = F;
34    using allocator_type = typename F::allocator_type;
35    using reference = V&;
36    using const_reference = V const&;
37    using size_type = std::size_t;
38    using self_type = cphstl::linked_nearly_complete_binary_tree<V, N
          , F>;
39    using navigator = cphstl::link_navigator<self_type>;
40    using const_navigator = cphstl::link_navigator<self_type const>;
41
42  private:
43
44    friend navigator;
45    friend const_navigator;
46
47    template <typename T, typename U, typename Some>
48    struct rebind;
49
50    template <typename T, typename U, template <typename...> class
          Name, typename... Args>
51    struct rebind<T, U, Name<Args...>> {
52      typedef Name<T, U> other;
53    };
54
55    using A = typename F::allocator_type;
56    using A_prime = typename A::template rebind<N>::other;
57    using F_prime = typename rebind<N, A_prime, F>::other;
58
59    size_type n;
60    N* root_access;
61    N* leaf_access;
62    N* leaf_dummy;
63    N* sibling_dummy;
64    F_prime factory;
65
66    template <typename M>
67    N* copy(M t) {
68      N* p = factory.create(*t);
69      if (t.is_leaf()) {
70        (*p).reset_right();
71        (*p).reset_left();
72        return p;
73      }
74      N* q = copy(t.left());
75      (*p).set_left(q);
76      (*p).reset_right();
77      if (t.has_right()) {
```

```
78          N* r = copy(t.right());
79          (*p).set_right(r);
80        }
81        return p;
82      }
83
84    public:
85
86      explicit linked_nearly_complete_binary_tree(F const& f = F())
87        : n(0), root_access(nullptr), leaf_access(nullptr),
88          leaf_dummy(nullptr), sibling_dummy(nullptr), factory(f) {
89        root_access = factory.create();
90        leaf_access = factory.create();
91        leaf_dummy = factory.create();
92        sibling_dummy = factory.create();
93        (*sibling_dummy).make_triangle(leaf_access, leaf_dummy);
94        (*leaf_access).set_left(leaf_access);
95        (*leaf_access).reset_right();
96        (*leaf_dummy).set_left(leaf_dummy);
97        (*leaf_dummy).reset_right();
98        (*sibling_dummy).set_left(sibling_dummy);
99        (*sibling_dummy).reset_right();
100       (*root_access).make_triangle(sibling_dummy, root_access);
101     }
102
103     template <typename S>
104     linked_nearly_complete_binary_tree(S const& s = S(), F const& f =
            F())
105       : n(0), root_access(nullptr), leaf_access(nullptr), leaf_dummy(
            nullptr),
106         sibling_dummy(nullptr), factory(f) {
107       root_access = factory.create();
108       leaf_access = factory.create();
109       leaf_dummy = factory.create();
110       sibling_dummy = factory.create();
111       (*sibling_dummy).make_triangle(leaf_access, leaf_dummy);
112       (*leaf_access).set_left(leaf_access);
113       (*leaf_access).reset_right();
114       (*leaf_dummy).set_left(leaf_dummy);
115       (*leaf_dummy).reset_right();
116       (*sibling_dummy).set_left(sibling_dummy);
117       (*sibling_dummy).reset_right();
118       (*root_access).make_triangle(sibling_dummy, root_access);
119       try {
120         for (size_type i = 0; i != s.size(); ++i) {
121           navigator leaf = expand();
122           *leaf = s[i];
123         }
124       }
125       catch (...) {
126         clear();
127         throw;
```

```
128       }
129     }
130
131     linked_nearly_complete_binary_tree(
            linked_nearly_complete_binary_tree const& other)
132       : n(other.size()), root_access(nullptr), leaf_access(nullptr),
133         leaf_dummy(nullptr), sibling_dummy(nullptr),
134         factory(other.get_allocator()) {
135       root_access = factory.create();
136       leaf_access = factory.create();
137       leaf_dummy = factory.create();
138       sibling_dummy = factory.create();
139       (*sibling_dummy).make_triangle(leaf_access, leaf_dummy);
140       (*leaf_access).set_left(leaf_access);
141       (*leaf_access).reset_right();
142       (*leaf_dummy).set_left(leaf_dummy);
143       (*leaf_dummy).reset_right();
144       (*sibling_dummy).set_left(sibling_dummy);
145       (*sibling_dummy).reset_right();
146       (*root_access).make_triangle(sibling_dummy, root_access);
147       if (n == 0) {
148         return;
149       }
150       N* p = copy(other.root());
151       (*root_access).make_triangle(p, root_access);
152       int const h = ilogb(n);
153       size_type mask = 1 << h;
154       for (int i = 0; i != h; ++i) {
155         mask >>= 1;
156         if ((n bitand mask) == 0) {
157           p = (*p).left();
158         }
159         else {
160           p = (*p).right();
161         }
162       }
163       (*p).make_triangle(leaf_access, leaf_dummy);
164     }
165
166     linked_nearly_complete_binary_tree(
            linked_nearly_complete_binary_tree&& other)
167       : n(0), root_access(nullptr), leaf_access(nullptr), leaf_dummy(
              nullptr),
168         sibling_dummy(nullptr), factory(other.factory) {
169       root_access = factory.create();
170       leaf_access = factory.create();
171       leaf_dummy = factory.create();
172       sibling_dummy = factory.create();
173       (*sibling_dummy).make_triangle(leaf_access, leaf_dummy);
174       (*leaf_access).set_left(leaf_access);
175       (*leaf_access).reset_right();
176       (*leaf_dummy).set_left(leaf_dummy);
```

```
177       (*leaf_dummy).reset_right();
178       (*sibling_dummy).set_left(sibling_dummy);
179       (*sibling_dummy).reset_right();
180       (*root_access).make_triangle(sibling_dummy, root_access);
181       swap(other);
182     }
183
184     ~linked_nearly_complete_binary_tree() {
185       clear();
186       factory.destroy(root_access);
187       factory.destroy(leaf_access);
188       factory.destroy(leaf_dummy);
189       factory.destroy(sibling_dummy);
190     }
191
192     template <typename T>
193     linked_nearly_complete_binary_tree& operator=(T const& other) {
194       linked_nearly_complete_binary_tree tmp(other);
195       clear();
196       swap(tmp);
197       return *this;
198     }
199
200     linked_nearly_complete_binary_tree& operator=(
          linked_nearly_complete_binary_tree&& other) {
201       linked_nearly_complete_binary_tree tmp(std::move(other));
202       clear();
203       swap(tmp);
204       return *this;
205     }
206
207     size_type size() const {
208       return n;
209     }
210
211     const_navigator root() const {
212       return const_navigator((*root_access).left());
213     }
214
215     navigator root() {
216       return navigator((*root_access).left());
217     }
218
219     const_navigator last() const {
220       return const_navigator((*leaf_access).parent());
221     }
222
223     navigator last() {
224       return navigator((*leaf_access).parent());
225     }
226
227     navigator expand() {
```

```
228       N* p = factory.create();
229       return attach(navigator(p));
230     }
231
232     navigator attach(navigator outsider) {
233       N* q; // parent of the new last leaf
234       N* p = outsider.p;
235       n += 1;
236       int const h = __builtin_ctzl(n);
237       N* const ℓ = (*leaf_access).parent();
238       if (__builtin_popcountl(n) == 1) {
239         if (n == 1) {
240           (*root_access).make_triangle(p, root_access);
241         }
242         else {
243           q = root_access;
244           for (int i = 0; i != h; ++i) {
245             q = (*q).left();
246           }
247           (*ℓ).reset_right();
248           (*ℓ).reset_left();
249           (*q).make_triangle(p, sibling_dummy);
250         }
251       }
252       else if ((n bitand 1) == 0) {
253         q = ℓ;
254         for (int i = 0; i != h + 1; ++i) {
255           q = (*q).parent();
256         }
257         q = (*q).right();
258         for (int i = 1; i < h; ++i) {
259           q = (*q).left();
260         }
261         (*ℓ).reset_right();
262         (*ℓ).reset_left();
263         (*q).make_triangle(p, sibling_dummy);
264       }
265       else {
266         q = (*ℓ).parent();
267         (*ℓ).reset_right();
268         (*ℓ).reset_left();
269         (*q).set_right(p);
270       }
271       (*p).make_triangle(leaf_access, leaf_dummy);
272       return navigator(p);
273     }
274
275     navigator detach() {
276       N* q; // new last leaf
277       int const h = __builtin_ctzl(n);
278       N* const ℓ = (*leaf_access).parent();
279       if (__builtin_popcountl(n) == 1) {
```

```
280         if (n == 1) {
281           q = sibling_dummy;
282         }
283         else {
284           q = (*root_access).left();
285           for (int i = 1; i != h; ++i) {
286             q = (*q).right();
287           }
288           N* p = (*ℓ).parent();
289           (*p).reset_right();
290           (*p).reset_left();
291         }
292       }
293       else if ((n bitand 1) == 0) {
294         q = ℓ;
295         for (int i = 0; i != h + 1; ++i) {
296           q = (*q).parent();
297         }
298         q = (*q).left();
299         for (int i = 0; i != h; ++i) {
300           q = (*q).right();
301         }
302         N* p = (*ℓ).parent();
303         (*p).reset_right();
304         (*p).reset_left();
305       }
306       else {
307         N* p = (*ℓ).parent();
308         q = (*p).left();
309         (*p).make_triangle(q, sibling_dummy);
310       }
311       (*q).make_triangle(leaf_access, leaf_dummy);
312       n -= 1;
313       return navigator(ℓ);
314     }
315
316     void remove(navigator surplus) {
317       factory.destroy(surplus.p);
318     }
319
320     void contract() {
321       navigator surplus = detach();
322       remove(surplus);
323     }
324
325     void swap(linked_nearly_complete_binary_tree& other) {
326       std::swap(n, other.n);
327       std::swap(root_access, other.root_access);
328       std::swap(leaf_access, other.leaf_access);
329       std::swap(leaf_dummy, other.leaf_dummy);
330       std::swap(sibling_dummy, other.sibling_dummy);
331       factory.swap(other.factory);
```

```
332     }
333
334     void clear() {
335       while (size() ≠ 0) {
336         contract();
337       }
338     }
339   };
340 }
341
342 #endif
```

Tuned-linked/link_navigator.h++

```
1  /*
2    A link navigator encapsulates a position of a value by storing a
3    pointer to the node that stores it; all the functionality is
4    provided by the node
5
6    Version: Tuned
7
8    Author: Jyrki Katajainen © 2015
9  */
10
11 #ifndef __CPHSTL_LINK_NAVIGATOR__
12 #define __CPHSTL_LINK_NAVIGATOR__
13 #include <limits> // std::numeric_limits
14 #include <sstream> // std::stringstream
15 #include <type_traits> // std::conditional, std::is_const, std::
       remove_const
16
17 namespace cphstl {
18
19   template <typename T>
20   class link_navigator {
21
22   public:
23
24     // type aliases
25
26     using owner_type = T;
27
28   private:
29
30     using T_bar = typename std::conditional<std::is_const<T>::value,
         typename std::remove_const<T>::type, T const>::type;
31     using L = link_navigator<T>;
32     using L_bar = link_navigator<T_bar>;
33     using V = typename T::value_type;
34     using N = typename T::node_type;
```

```
35      using V_check = typename std::conditional<std::is_const<T>::value
            , V const, V>::type;
36      using pointer = V_check*;
37      using reference = V_check&;
38
39      // friends
40
41      friend T;
42      friend L_bar;
43
44      // variables
45
46      N* p;
47
48      // parameterized constructor
49
50      link_navigator(N* q)
51        : p(q) {
52      }
53
54    public:
55
56      // default constructor
57
58      link_navigator()
59        : p(nullptr) {
60      }
61
62      // copy constructors
63
64      // Generated by the compiler if needed
65      // link_navigator(const link_navigator&) = default;
66
67      link_navigator(link_navigator<typename std::remove_const<T>::type
            > const& x)
68        : p(x.p) {
69      }
70
71      // assignments
72
73      // Generated by the compiler if needed
74      // link_navigator& operator=(link_navigator const&) = default;
75
76      link_navigator& operator=(link_navigator<typename std::
            remove_const<T>::type> const& x) {
77        p = x.p;
78        return *this;
79      }
80
81      // destructor
82
83      ~link_navigator() {
```

```
84      };
85
86      // operator*
87
88      reference operator*() const {
89        return (*p).value();
90      }
91
92      // operator->
93
94      pointer operator->() const {
95        return &(*p).value();
96      }
97
98      // left
99
100     L left() const {
101       return L((*p).left());
102     }
103
104     // right
105
106     L right() const {
107       return L((*p).right());
108     }
109
110     // parent
111
112     L parent() const {
113       return L((*p).parent());
114     }
115
116     // slide
117
118     void slide(L const& neighbour) {
119       N* q = neighbour.p;
120       N* o = (*p).parent();
121       N* r = (*q).parent();
122       bool p_on_left = ((*o).left() == p);
123       bool q_on_left = ((*r).left() == q);
124       if (o == q) {
125         if (q_on_left) {
126           if (p_on_left) {
127             left_left_case(r, q, p);
128           }
129           else {
130             left_right_case(r, q, p);
131           }
132         }
133         else {
134           if (p_on_left) {
135             right_left_case(r, q, p);
```

```
136           }
137           else {
138             right_right_case(r, q, p);
139           }
140         }
141       }
142       else {
143         if (p_on_left) {
144           if (q_on_left) {
145             left_left_case(o, p, q);
146           }
147           else {
148             left_right_case(o, p, q);
149           }
150         }
151         else {
152           if (q_on_left) {
153             right_left_case(o, p, q);
154           }
155           else {
156             right_right_case(o, p, q);
157           }
158         }
159       }
160     }
161
162     // replace
163
164     void replace(L const& replacement) {
165       N* q = replacement.p;
166       N* o = (*p).parent();
167       N* t = (*p).left();
168       bool p_on_left = ((*o).left() == p);
169       if (p_on_left) {
170         N* r = (*o).right();
171         (*q).set_left_corner(o, r);
172         if (t ≠ nullptr) {
173           N* u = (*p).right();
174           (*q).set_top_corner(t, u);
175         }
176         else {
177           (*q).reset_left();
178         }
179       }
180       else {
181         N* r = (*o).left();
182         (*q).set_right_corner(o, r);
183         if (t ≠ nullptr) {
184           N* u = (*p).right();
185           (*q).set_top_corner(t, u);
186         }
187         else {
```

```
188           (*q).reset_left();
189         }
190       }
191     }
192
193     // operator==
194
195     bool operator==(L const& other) const {
196       return (p == other.p);
197     }
198
199     bool operator==(L_bar const& other) const {
200       return (p == other.p);
201     }
202
203     // operator≠
204
205     bool operator≠(L const& other) const {
206       return not (*this == other);
207     }
208
209     bool operator≠(L_bar const& other) const {
210       return not (*this == other);
211     }
212
213     // is_root
214
215     bool is_root() const {
216       return (*p).is_root();
217     }
218
219     // is_leaf
220
221     bool is_leaf() const {
222       N* below = (*p).left();
223       return (below == nullptr) or ((*below).left() == below);
224     }
225
226     // has_right
227
228     bool has_right() const {
229       N* below = (*p).left();
230       if (below == nullptr) {
231         return false;
232       }
233       if ((*below).left() == below) {
234         return false;
235       }
236       below= (*p).right();
237       if (below == nullptr) {
238         return false;
239       }
```

```
240       if ((*below).left() == below) {
241         return false;
242       }
243       return true;
244     }
245
246     // print
247
248     void print(std::ostream& stream) const {
249       if (std::is_const<L>::value) {
250         stream << "const navigator: node at " << std::hex << p;
251       }
252       else {
253         stream << "navigator: node at " << std::hex << p;
254       }
255     }
256
257   private:
258
259     // swap p and q, o: parent of p
260     void left_left_case(N* o, N* p, N* q) {
261       N* r = (*o).right();
262       N* s = (*p).right();
263       N* t = (*q).left();
264       if (t ≠ nullptr) {
265         N* u = (*q).right();
266         (*p).set_top_corner(t, u);
267       }
268       else {
269         (*p).reset_left();
270       }
271       (*q).make_triangle(p, s);
272       (*q).set_left_corner(o, r);
273     }
274
275     void left_right_case(N* o, N* p, N* q) {
276       N* r = (*o).right();
277       N* s = (*p).left();
278       N* t = (*q).left();
279       if (t ≠ nullptr) {
280         N* u = (*q).right();
281         (*p).set_top_corner(t, u);
282       }
283       else {
284         (*p).reset_left();
285       }
286       (*q).make_triangle(s, p);
287       (*q).set_left_corner(o, r);
288     }
289
290     void right_left_case(N* o, N* p, N* q) {
291       N* r = (*o).left();
```

```
292       N* s = (*p).right();
293       N* t = (*q).left();
294       if (t != nullptr) {
295         N* u = (*q).right();
296         (*p).set_top_corner(t, u);
297       }
298       else {
299         (*p).reset_left();
300       }
301       (*q).make_triangle(p, s);
302       (*q).set_right_corner(o, r);
303     }
304
305     void right_right_case(N* o, N* p, N* q) {
306       N* r = (*o).left();
307       N* s = (*p).left();
308       N* t = (*q).left();
309       if (t != nullptr) {
310         N* u = (*q).right();
311         (*p).set_top_corner(t, u);
312       }
313       else {
314         (*p).reset_left();
315       }
316       (*q).make_triangle(s, p);
317       (*q).set_right_corner(o, r);
318     }
319   };
320
321   template <typename T>
322   std::ostream& operator<<(std::ostream& stream, link_navigator<T>
          const& t) {
323     t.print(stream);
324     return stream;
325   }
326 }
327
328 #endif
```

Tuned-linked/binary_tree_node.h++

```
1  /*
2    A node used in any binary tree
3
4    Version: Tuned
5
6    Author: Jyrki Katajainen © 2015
7  */
8
9  #ifndef __CPHSTL_BINARY_TREE_NODE__
10 #define __CPHSTL_BINARY_TREE_NODE__
```

```
11
12 #include <utility> // std::move
13
14 namespace cphstl {
15
16   template <typename V>
17   class binary_tree_node {
18   public:
19
20     using value_type = V;
21
22     explicit binary_tree_node()
23       : element(), first(nullptr), second(nullptr), third(nullptr) {
24     }
25
26     binary_tree_node(V const& v)
27       : element(v), first(nullptr), second(nullptr), third(nullptr) {
28     }
29
30     binary_tree_node(V&& v)
31       : element(std::move(v)), first(nullptr), second(nullptr), third
             (nullptr) {
32     }
33
34     bool is_root() const {
35       return (*third).third == third;
36     }
37
38     V const& value() const {
39       return element;
40     }
41
42     binary_tree_node* left() const {
43       return first;
44     }
45
46     binary_tree_node* right() const {
47       return second;
48     }
49
50     binary_tree_node* parent() const {
51       return third;
52     }
53
54     V& value() {
55       return element;
56     }
57
58     void set_left(binary_tree_node* q) {
59       first = q;
60       (*q).third = this;
61     }
```

```
62
63      void reset_left() {
64        first = nullptr;
65      }
66
67      void set_right(binary_tree_node* q) {
68        second = q;
69        (*q).third = this;
70      }
71
72      void reset_right() {
73        second = nullptr;
74      }
75
76      void make_triangle(binary_tree_node* q, binary_tree_node* r) {
77        first = q;
78        second = r;
79        (*q).third = this;
80        (*r).third = this;
81      }
82
83      void set_top_corner(binary_tree_node* q, binary_tree_node* r) {
84        first = q;
85        second = r;
86        (*q).third = this;
87        (*r).third = this;
88      }
89
90      void set_left_corner(binary_tree_node* p, binary_tree_node*) {
91        third = p;
92        (*p).first = this;
93      }
94
95      void set_right_corner(binary_tree_node* p, binary_tree_node*) {
96        third = p;
97        (*p).second = this;
98      }
99
100   private:
101
102     V element;
103     binary_tree_node* first;
104     binary_tree_node* second;
105     binary_tree_node* third;
106   };
107 }
108
109 #endif
```

Tuned-linked/compact_binary_tree_node.h++

```
/*
  A node used in compact binary trees

  Version: Tuned

  Author: Jyrki Katajainen © 2015
*/

#ifndef __CPHSTL_COMPACT_BINARY_TREE_NODE__
#define __CPHSTL_COMPACT_BINARY_TREE_NODE__

#include <utility> // std::move

namespace cphstl {

  template <typename V>
  class compact_binary_tree_node {
  public:

    using value_type = V;

    explicit compact_binary_tree_node()
      : element(), first(nullptr), second(nullptr) {
    }

    compact_binary_tree_node(V const& v)
      : element(v), first(nullptr), second(nullptr) {
    }

    compact_binary_tree_node(V&& v)
      : element(std::move(v)), first(nullptr), second(nullptr) {
    }

    bool is_root() const {
      return (*second).second == second;
    }

    V const& value() const {
      return element;
    }

    compact_binary_tree_node* left() const {
      return first;
    }

    compact_binary_tree_node* right() const {
      return first -> second;
    }

    compact_binary_tree_node* parent() const {
      bool on_left = (second -> second -> first == this);
      if (on_left) {
```

```
53          return second → second;
54        }
55        return second;
56      }
57
58      V& value() {
59        return element;
60      }
61
62      void set_left(compact_binary_tree_node* q) {
63        first = q;
64      }
65
66      void reset_left() {
67        first = nullptr;
68      }
69
70      void set_right(compact_binary_tree_node* q) {
71        (*first).second = q;
72        (*q).second = this;
73      }
74
75      void reset_right() {
76        (*first).second = this;
77      }
78
79      void make_triangle(compact_binary_tree_node* q,
            compact_binary_tree_node* r) {
80        first = q;
81        (*q).second = r;
82        (*r).second = this;
83      }
84
85      void set_top_corner(compact_binary_tree_node* q,
            compact_binary_tree_node* r) {
86        first = q;
87        (*r).second = this;
88      }
89
90      void set_left_corner(compact_binary_tree_node* p,
            compact_binary_tree_node* r) {
91        (*p).first = this;
92        second = r;
93      }
94
95      void set_right_corner(compact_binary_tree_node* p,
            compact_binary_tree_node* q) {
96        (*q).second = this;
97        second = p;
98      }
99
100   private:
```

```
101
102     V element;
103     compact_binary_tree_node* first;
104     compact_binary_tree_node* second;
105   };
106 }
107
108 #endif
```

Tuned-linked/node_factory.h++

This file is a symbolic link to Referent/node_factory.h++.

Tuned-linked/binary_heap.i++

This file is a symbolic link to Linked/binary_heap.i++.

Drivers

Drivers/construct-driver.c++

```
1  #if not defined(MAXSIZE)
2  #define MAXSIZE (32 * 1024 * 1024)
3  #endif
4
5  #include <algorithm> // std::max
6  #include <cstdlib> // std::rand
7  #include <ctime> // std::clock_t, std::clock, CLOCKS_PER_SEC
8  #include <iostream> // std::cout
9  #include <iterator> // std::iterator_traits
10 #include <memory> // std::allocator
11 #include <new> // std placement new
12 #include <vector> // std::vector
13 #include <utility> // std::move, std::swap
14
15 #ifdef PROFILING
16
17 #include <callgrind.h>
18
19 #endif
20
21 long volatile comparisons = 0;
22 long volatile assignments = 0;
23
24 #include "data_structure.i++" // Q comes from here
25 #include <iterator> // defines std::iterator_traits
26
27 template <typename I>
```

```
28 void increasing_sequence(I first, I past_the_end) {
29   using V = typename std::iterator_traits<I>::value_type;
30   for (I q = first; q ≠ past_the_end; ++q) {
31     *q = V(unsigned(q − first));
32   }
33 }
34
35 template <typename I>
36 void decreasing_sequence(I first, I past_the_end) {
37   using V = typename std::iterator_traits<I>::value_type;
38   for (I q = first; q ≠ past_the_end; ++q) {
39     *q = V(unsigned(past_the_end − q));
40   }
41 }
42
43 template <typename I>
44 void random_sequence(I first, I past_the_end) {
45   using V = typename std::iterator_traits<I>::value_type;
46   for (I q = first; q ≠ past_the_end; ++q) {
47     *q = V(unsigned(std::rand()));
48   }
49 }
50
51 template <typename I>
52 void random_permutation(I first, I past_the_end) {
53   using V = typename std::iterator_traits<I>::value_type;
54   int n = past_the_end − first;
55   for (int i = 0; i ≠ n; ++i) {
56     *(first + i) = V(i);
57   }
58   srand(1);
59   for (int i = 0; i ≠ n; ++i) {
60     std::swap(*(first + i), *(first + (rand() % (n))));
61   }
62 }
63
64 template <typename I>
65 void zero_sequence(I first, I past_the_end) {
66   typedef typename std::iterator_traits<I>::value_type V;
67   I q;
68   for (q = first; q ≠ past_the_end; ++q) {
69     *q = V(unsigned(0));
70   }
71 }
72
73 template <typename I>
74 void binary_sequence(I first, I past_the_end) {
75   typedef typename std::iterator_traits<I>::value_type V;
76   I q;
77   for (q = first; q ≠ past_the_end; ++q) {
78     *q = V(unsigned(std::rand()) & 1);
79   }
```

```
80 }
81
82 int main() {
83   using A = std::allocator<V>;
84   using D = std::vector<V, A>;
85   int const n = NUMBER;
86   unsigned int const repetitions = std::max(2, MAXSIZE / n);
87
88   D* sequence = new D[repetitions];
89   for (volatile unsigned int t = 0; t != repetitions; ++t)  {
90     sequence[t].resize(n);
91     DATA(sequence[t].begin(), sequence[t].end());
92   }
93
94   comparisons = 0;
95   assignments = 0;
96   Q* heap = new Q[repetitions];
97
98   std::clock_t start = std::clock();
99
100 #ifdef PROFILING
101
102   CALLGRIND_START_INSTRUMENTATION;
103
104 #endif
105
106   for (volatile unsigned int t = 0; t != repetitions; ++t)  {
107     heap[t].~Q();
108     new (&heap[t]) Q(C(), std::move(sequence[t]));
109   }
110
111 #ifdef PROFILING
112
113   CALLGRIND_STOP_INSTRUMENTATION;
114
115 #endif
116
117   std::clock_t stop = std::clock();
118
119   if (comparisons != 0) {
120     double comparisons_per_run = double(comparisons) / double(
          repetitions);
121     double comparisons_per_element = comparisons_per_run / double(n);
122     std::cout.precision(3);
123     std::cout << n << " " << comparisons_per_element << "\n";
124   }
125   else if (assignments != 0) {
126     double assignments_per_run = double(assignments) / double(
          repetitions);
127     double assignments_per_element = assignments_per_run / double(n);
128     std::cout.precision(3);
129     std::cout << n << " " << assignments_per_element << "\n";
```

```
130   }
131   else {
132     double running_time = double(stop - start)/double(CLOCKS_PER_SEC)
          ;
133     double time_per_run = 1000000000.0 * running_time / double(
          repetitions);
134     double time_per_element = time_per_run / double(n);
135     std::cout.precision(4);
136     std::cout << n << " " << time_per_element << "\n";
137   }
138   delete[] sequence;
139   delete[] heap;
140   return 0;
141 }
```

Drivers/push-driver.c++

```
1  #if not defined(MAXSIZE)
2  #define MAXSIZE (32 * 1024 * 1024)
3  #endif
4
5  #include <algorithm> // std::max
6  #include <cstdlib> // std::rand
7  #include <ctime> // std::clock_t, std::clock, CLOCKS_PER_SEC
8  #include <iostream> // std::cout
9  #include <iterator> // std::iterator_traits
10 #include <memory> // std::allocator
11 #include <vector> // std::vector
12 #include <utility> // std::move, std::swap
13
14 #ifdef PROFILING
15
16 #include <callgrind.h>
17
18 #endif
19
20 long volatile comparisons = 0;
21 long volatile assignments = 0;
22
23 #include "data_structure.i++" // Q comes from here
24 #include <iterator> // defines std::iterator_traits
25
26 template <typename I>
27 void increasing_sequence(I first, I past_the_end) {
28   using V = typename std::iterator_traits<I>::value_type;
29   for (I q = first; q != past_the_end; ++q) {
30     *q = std::move(V(unsigned(q - first)));
31   }
32 }
33
34 template <typename I>
```

```
35 void decreasing_sequence(I first, I past_the_end) {
36   using V = typename std::iterator_traits<I>::value_type;
37   for (I q = first; q ≠ past_the_end; ++q) {
38     *q = std::move(V(unsigned(past_the_end − q)));
39   }
40 }
41
42 template <typename I>
43 void random_sequence(I first, I past_the_end) {
44   using V = typename std::iterator_traits<I>::value_type;
45   for (I q = first; q ≠ past_the_end; ++q) {
46     *q = std::move(V(unsigned(std::rand())));
47   }
48 }
49
50 template <typename I>
51 void random_permutation(I first, I past_the_end) {
52   using V = typename std::iterator_traits<I>::value_type;
53   int n = past_the_end − first;
54   for (int i = 0; i ≠ n; ++i) {
55     first[i] = std::move(V(i));
56   }
57   srand(1);
58   for (int i = 0; i ≠ n; ++i) {
59     std::swap(first[i], first[rand() % (n)]);
60   }
61 }
62
63 template <typename I>
64 void zero_sequence(I first, I past_the_end) {
65   typedef typename std::iterator_traits<I>::value_type V;
66   I q;
67   for (q = first; q ≠ past_the_end; ++q) {
68     *q = std::move(V(unsigned(0)));
69   }
70 }
71
72 template <typename I>
73 void binary_sequence(I first, I past_the_end) {
74   typedef typename std::iterator_traits<I>::value_type V;
75   I q;
76   for (q = first; q ≠ past_the_end; ++q) {
77     *q = std::move(V(unsigned(std::rand()) & 1));
78   }
79 }
80
81 int main() {
82   using V = Q::value_type;
83   using D = std::vector<V, std::allocator<V>>;
84
85   int const n = NUMBER;
86   unsigned int const repetitions = std::max(2, MAXSIZE / n);
```

```
87
88    D* sequence = new D[repetitions];
89    for (volatile unsigned int t = 0; t != repetitions; ++t)  {
90      sequence[t].resize(n);
91      DATA(sequence[t].begin(), sequence[t].end());
92    }
93
94    Q* many = new Q[repetitions];
95    comparisons = 0;
96    assignments = 0;
97    std::clock_t start = std::clock();
98
99  #ifdef PROFILING
100
101   CALLGRIND_START_INSTRUMENTATION;
102
103 #endif
104
105   for (volatile unsigned int t = 0; t != repetitions; ++t)  {
106     for (int i = 0; i != n; ++i) {
107       many[t].push(sequence[t][i]);
108     }
109   }
110
111 #ifdef PROFILING
112
113   CALLGRIND_STOP_INSTRUMENTATION;
114
115 #endif
116
117   std::clock_t stop = std::clock();
118
119   if (comparisons != 0) {
120     double comparisons_per_run = double(comparisons) / double(
          repetitions);
121     double comparisons_per_operation = comparisons_per_run / double(n
          );
122     std::cout.precision(3);
123     std::cout << n << " " << comparisons_per_operation << "\n";
124   }
125   else if (assignments != 0) {
126     double assignments_per_run = double(assignments) / double(
          repetitions);
127     double assignments_per_operation = assignments_per_run / double(n
          );
128     std::cout.precision(3);
129     std::cout << n << " " << assignments_per_operation << "\n";
130   }
131   else {
132     double running_time = double(stop - start)/double(CLOCKS_PER_SEC)
          ;
```

```
133     double time_per_run = 1000000000.0 * running_time / double(
            repetitions);
134     double time_per_operation = time_per_run / double(n);
135     std::cout.precision(4);
136     std::cout << n << " " << time_per_operation << "\n";
137   }
138
139   delete[] many;
140   delete[] sequence;
141   return 0;
142 }
```

Drivers/pop-driver.c++

```
1  #if not defined(MAXSIZE)
2  #define MAXSIZE (32 * 1024 * 1024)
3  #endif
4
5  #include <algorithm> // std::max
6  #include <cstdlib> // std::rand
7  #include <ctime> // std::clock_t, std::clock, CLOCKS_PER_SEC
8  #include <iostream> // std::cout
9  #include <iterator> // std::iterator_traits
10 #include <memory> // std::allocator
11 #include <vector> // std::vector
12 #include <utility> // std::swap
13
14 #ifdef PROFILING
15
16 #include <callgrind.h>
17
18 #endif
19
20 long volatile comparisons = 0;
21 long volatile assignments = 0;
22
23 #include "data_structure.i++" // Q comes from here
24 #include <iterator> // defines std::iterator_traits
25
26 template <typename I>
27 void increasing_sequence(I first, I past_the_end) {
28   using V = typename std::iterator_traits<I>::value_type;
29   for (I q = first; q != past_the_end; ++q) {
30     *q = std::move(V(unsigned(q - first)));
31   }
32 }
33
34 template <typename I>
35 void decreasing_sequence(I first, I past_the_end) {
36   using V = typename std::iterator_traits<I>::value_type;
37   for (I q = first; q != past_the_end; ++q) {
```

```
38      *q = std::move(V(unsigned(past_the_end - q)));
39    }
40  }
41
42  template <typename I>
43  void random_sequence(I first, I past_the_end) {
44    using V = typename std::iterator_traits<I>::value_type;
45    for (I q = first; q != past_the_end; ++q) {
46      *q = std::move(V(unsigned(std::rand())));
47    }
48  }
49
50  template <typename I>
51  void random_permutation(I first, I past_the_end) {
52    using V = typename std::iterator_traits<I>::value_type;
53    int n = past_the_end - first;
54    for (int i = 0; i != n; ++i) {
55      *(first + i) = std::move(V(std::move(i)));
56    }
57    srand(1);
58    for (int i = 0; i != n; ++i) {
59      std::swap(*(first + i), *(first + (rand() % (n))));
60    }
61  }
62
63  template <typename I>
64  void zero_sequence(I first, I past_the_end) {
65    typedef typename std::iterator_traits<I>::value_type V;
66    I q;
67    for (q = first; q != past_the_end; ++q) {
68      *q = V(unsigned(0));
69    }
70  }
71
72  template <typename I>
73  void binary_sequence(I first, I past_the_end) {
74    typedef typename std::iterator_traits<I>::value_type V;
75    I q;
76    for (q = first; q != past_the_end; ++q) {
77      *q = V(unsigned(std::rand()) & 1);
78    }
79  }
80
81  int main() {
82    using V = Q::value_type;
83    using D = std::vector<V, std::allocator<V>>;
84
85    int const n = NUMBER;
86    unsigned int const repetitions = std::max(2, MAXSIZE / n);
87
88    D* sequence = new D[repetitions];
89    for (volatile unsigned int t = 0; t != repetitions; ++t)  {
```

```
90      sequence[t].resize(n);
91      DATA(sequence[t].begin(), sequence[t].end());
92    }
93
94    Q* many = new Q[repetitions];
95    for (volatile unsigned int t = 0; t != repetitions; ++t) {
96      for (int i = 0; i != n; ++i) {
97        many[t].push(sequence[t][i]);
98      }
99    }
100
101   comparisons = 0;
102   assignments = 0;
103
104   std::clock_t start = std::clock();
105
106 #ifdef PROFILING
107
108   CALLGRIND_START_INSTRUMENTATION;
109
110 #endif
111
112   for (volatile unsigned int t = 0; t != repetitions; ++t) {
113     for (int i = 0; i != n; ++i) {
114       many[t].pop();
115     }
116   }
117
118 #ifdef PROFILING
119
120   CALLGRIND_STOP_INSTRUMENTATION;
121
122 #endif
123
124   std::clock_t stop = std::clock();
125
126   if (comparisons != 0) {
127     double comparisons_per_run = double(comparisons) / double(
          repetitions);
128     double comparisons_per_operation = comparisons_per_run / double(n
          );
129     std::cout.precision(3);
130     std::cout << n << " " << comparisons_per_operation << "\n";
131   }
132   else if (assignments != 0) {
133     double assignments_per_run = double(assignments) / double(
          repetitions);
134     double assignments_per_operation = assignments_per_run / double(n
          );
135     std::cout.precision(3);
136     std::cout << n << " " << assignments_per_operation << "\n";
137   }
```

```
138   else {
139     double running_time = double(stop - start)/double(CLOCKS_PER_SEC)
          ;
140     double time_per_run = 1000000000.0 * running_time / double(
          repetitions);
141     double time_per_operation = time_per_run / double(n);
142     std::cout.precision(4);
143     std::cout << n << " " << time_per_operation << "\n";
144   }
145
146   delete[] many;
147   delete[] sequence;
148   return 0;
149 }
```

Common

This directory is intensionally empty.

Aids

Aids/int.i++

```
1 using V = int;
```

Aids/counting_int.i++

```
1  #include <utility> // std::move
2
3  class counting_int {
4  private:
5
6    int datum;
7
8  public:
9
10   explicit counting_int(int x = 0)
11     : datum(std::move(x)) {
12     assignments += 1;
13   }
14
15   counting_int(counting_int const& other) {
16     datum = other.datum;
17     assignments += 1;
18   }
19
20   counting_int(counting_int&& other) {
21     datum = std::move(other.datum);
```

```
22      assignments += 1;
23    }
24
25    counting_int& operator=(counting_int const& other) {
26      datum = other.datum;
27      assignments += 1;
28      return *this;
29    }
30
31    counting_int& operator=(counting_int&& other) {
32      datum = std::move(other.datum);
33      assignments += 1;
34      return *this;
35    }
36
37    operator int() const {
38      return datum;
39    }
40
41    friend bool operator<(counting_int const&, counting_int const&);
42  };
43
44  bool operator<(counting_int const& x, counting_int const& y) {
45    return x.datum < y.datum;
46  }
47
48  using V = counting_int;
```

Aids/std_less.i++

```
1  #include <functional> // std::less
2
3  using C = std::less<V>;
4  C cmp = C();
```

Aids/counting_less.i++

```
1  #include <functional> // std::function
2
3  template <typename T>
4  class counting_comparator {
5  public:
6
7    typedef T first_argument_type;
8    typedef T second_argument_type;
9    typedef bool result_type;
10
11   bool operator()(T const& a, T const& b) const {
12     ++comparisons;
13     return a < b;
```

```
14     }
15 };
16
17 using C = counting_comparator<V>;
```

Aids/vector.i++

```
1 #include <memory> // std::allocator
2 #include <vector> // std::vector
3
4 using A = std::allocator<V>;
5 using S = std::vector<V, A>;
```

Aids/deque.i++

```
1 #include <memory> // std::allocator
2 #include <deque> // std::deque
3
4 using A = std::allocator<V>;
5 using S = std::deque<V, A>;
```

Aids/STANDARD.i++

```
1 #define STANDARD
```

Aids/COMPACT.i++

```
1 #define COMPACT
```

Makefile

benchmark.mk

```
1 # Usage example
2 # make -f benchmark.mk experiment=COMPACT Linked.time.push
3
4 CXXFLAGS = -DNDEBUG -Wall -std=c++11 -pedantic -x c++ -O3
5 CXX = g++
6
7 IFLAGS = -I. -ICommon/ -I$(HOME)/CPHSTL/Source/Priority-queue-
      frameworks/Code -I$(HOME)/CPHSTL/Source/Meldable-priority-queue/
      Code -I$(HOME)/CPHSTL/Source/Iterator/Code -I$(HOME)/CPHSTL/
      Source/Proxy/Code/ -I$(HOME)/CPHSTL/Source/Type/Code/
8
9 thisfile = benchmark.mk
10 size = 1024 32768 1048576 33554432
```

```
11 inc = increasing_sequence
12 ran = random_permutation
13 experiment = vector
14
15 default:
16
17 implementations:= Std Williams Jensen Edelkamp-Katajainen Goodrich-et
       -al Implicit Referent Inverse Linked Tuned-implicit Tuned-
       referent Tuned-inverse Tuned-linked
18 data-structures:= $(basename $(implementations))
19 time-tests = $(addsuffix .time, $(data-structures))
20 comp-tests = $(addsuffix .comp, $(data-structures))
21 move-tests = $(addsuffix .move, $(data-structures))
22 prof-tests = $(addsuffix .prof, $(data-structures))
23
24 construct-time-tests:= $(addsuffix .construct, $(time-tests))
25 construct-comp-tests:= $(addsuffix .construct, $(comp-tests))
26 construct-move-tests:= $(addsuffix .construct, $(move-tests))
27 construct-prof-tests:= $(addsuffix .construct, $(prof-tests))
28
29 push-time-tests:= $(addsuffix .push, $(time-tests))
30 push-comp-tests:= $(addsuffix .push, $(comp-tests))
31 push-move-tests:= $(addsuffix .push, $(move-tests))
32 push-prof-tests:= $(addsuffix .push, $(prof-tests))
33
34 pop-time-tests:= $(addsuffix .pop, $(time-tests))
35 pop-comp-tests:= $(addsuffix .pop, $(comp-tests))
36 pop-move-tests:= $(addsuffix .pop, $(move-tests))
37 pop-prof-tests:= $(addsuffix .pop, $(prof-tests))
38
39 $(time-tests): %.time: %.i++
40   make -s $*.time.construct
41   make -s $*.time.push
42   make -s $*.time.pop
43
44 $(comp-tests): %.comp: %.i++
45   make -s $*.comp.construct
46   make -s $*.comp.push
47   make -s $*.comp.pop
48
49 $(move-tests): %.move: %.i++
50   make -s $*.move.construct
51   make -s $*.move.push
52   make -s $*.move.pop
53
54 $(prof-tests): %.prof: %.i++
55   make -s $*.prof.construct
56   make -s $*.prof.push
57   make -s $*.prof.pop
58
59 $(construct-time-tests): %.time.construct : %/binary_heap.i++
60   @echo $*
```

```
61  @echo "construction: time per data element (int)"
62  @cp Aids/int.i++ element.i++
63  @cp Aids/std_less.i++ comparator.i++
64  @echo $(experiment)
65  @cp Aids/$(experiment).i++ experiment.i++
66  @cp $*/binary_heap.i++ data_structure.i++
67  @for d in $(ran) ; do \
68    echo $$d ; \
69    for x in $(size) ; do \
70      $(CXX) $(CXXFLAGS) $(IFLAGS) -I$* -DNUMBER=$$x -DDATA=$$d
          Drivers/construct_driver.c++ ; \
71      ./a.out ; \
72    done \
73  done
74  @make -s -f $(thisfile) clean
75
76 $(push_time_tests): %.time.push : %/binary_heap.i++
77  @echo $*
78  @echo "push: time per operation (int)"
79  @cp Aids/int.i++ element.i++
80  @cp Aids/std_less.i++ comparator.i++
81  @echo $(experiment)
82  @cp Aids/$(experiment).i++ experiment.i++
83  @cp $*/binary_heap.i++ data_structure.i++
84  @for d in $(inc) ; do \
85    echo $$d ; \
86    for x in $(size) ; do \
87      $(CXX) $(CXXFLAGS) $(IFLAGS) -I$* -DNUMBER=$$x -DDATA=$$d
          Drivers/push_driver.c++ ; \
88      ./a.out ; \
89    done \
90  done
91  @make -s -f $(thisfile) clean
92
93 $(pop_time_tests): %.time.pop : %/binary_heap.i++
94  @echo $*
95  @echo "pop: time per operation (int)"
96  @cp Aids/int.i++ element.i++
97  @cp Aids/std_less.i++ comparator.i++
98  @echo $(experiment)
99  @cp Aids/$(experiment).i++ experiment.i++
100 @cp $*/binary_heap.i++ data_structure.i++
101 @for d in $(ran) ; do\
102   echo $$d ; \
103   for x in $(size) ; do\
104     $(CXX) $(CXXFLAGS) $(IFLAGS) -I$* -DNUMBER=$$x -DDATA=$$d
          Drivers/pop_driver.c++ ; \
105     ./a.out ; \
106   done \
107 done
108 @make -s -f $(thisfile) clean
109
```

```
110 $(construct-comp-tests): %.comp.construct : %/binary_heap.i++
111   @echo $* "construction: #element comparisons per element"
112   @cp Aids/int.i++ element.i++
113   @cp Aids/counting_less.i++ comparator.i++
114   @cp Aids/$(experiment).i++ experiment.i++
115   @cp $*/binary_heap.i++ data-structure.i++
116   @for d in $(ran) ; do\
117     echo $$d ; \
118     for x in $(size) ; do\
119       $(CXX) $(CXXFLAGS) $(IFLAGS) -I$* -DNUMBER=$$x -DDATA=$$d
              Drivers/construct-driver.c++ ; \
120       ./a.out ; \
121     done \
122   done
123   @make -s -f $(thisfile) clean
124
125 $(push-comp-tests): %.comp.push : %/binary_heap.i++
126   @echo $* "push: #element comparisons per operation"
127   @cp Aids/int.i++ element.i++
128   @cp Aids/counting_less.i++ comparator.i++
129   @cp Aids/$(experiment).i++ experiment.i++
130   @cp $*/binary_heap.i++ data-structure.i++
131   @for d in $(inc) ; do\
132     echo $$d ; \
133     for x in $(size) ; do\
134       $(CXX) $(CXXFLAGS) $(IFLAGS) -I$* -DNUMBER=$$x -DDATA=$$d
              Drivers/push-driver.c++ ; \
135       ./a.out ; \
136     done \
137   done
138   @make -s -f $(thisfile) clean
139
140 $(pop-comp-tests): %.comp.pop : %/binary_heap.i++
141   @echo $* "pop: #element comparisons per operation"
142   @cp Aids/int.i++ element.i++
143   @cp Aids/counting_less.i++ comparator.i++
144   @cp Aids/$(experiment).i++ experiment.i++
145   @cp $*/binary_heap.i++ data-structure.i++
146   @for d in $(ran) ; do\
147     echo $$d ; \
148     for x in $(size) ; do\
149       $(CXX) $(CXXFLAGS) $(IFLAGS) -I$* -DNUMBER=$$x -DDATA=$$d -
              DDATA=$$d Drivers/pop-driver.c++ ; \
150       ./a.out ; \
151     done \
152   done
153   @make -s -f $(thisfile) clean
154
155 $(construct-move-tests): %.move.construct : %/binary_heap.i++
156   @echo $* "construction: #element assignments per element"
157   @cp Aids/counting_int.i++ element.i++
158   @cp Aids/std_less.i++ comparator.i++
```

```
159   @cp Aids/$(experiment).i++ experiment.i++
160   @cp $*/binary_heap.i++ data_structure.i++
161   @for d in $(ran) ; do \
162     echo $$d ; \
163     for x in $(size) ; do \
164       $(CXX) $(CXXFLAGS) $(IFLAGS) -I$* -DNUMBER=$$x -DDATA=$$d
              Drivers/construct_driver.c++ ; \
165       ./a.out ; \
166     done \
167   done
168   @make -s -f $(thisfile) clean
169
170 $(push_move_tests): %.move.push : %/binary_heap.i++
171   @echo $* "push: #element assignments per operation"
172   @cp Aids/counting_int.i++ element.i++
173   @cp Aids/std_less.i++ comparator.i++
174   @cp Aids/$(experiment).i++ experiment.i++
175   @cp $*/binary_heap.i++ data_structure.i++
176   @for d in $(inc) ; do \
177     echo $$d ; \
178     for x in $(size) ; do \
179       $(CXX) $(CXXFLAGS) $(IFLAGS) -I$* -DNUMBER=$$x -DDATA=$$d
              Drivers/push_driver.c++ ; \
180       ./a.out ; \
181     done \
182   done
183   @make -s -f $(thisfile) clean
184
185 $(pop_move_tests): %.move.pop : %/binary_heap.i++
186   @echo $* "pop: #element assignments per operation"
187   @cp Aids/counting_int.i++ element.i++
188   @cp Aids/std_less.i++ comparator.i++
189   @cp Aids/$(experiment).i++ experiment.i++
190   @cp $*/binary_heap.i++ data_structure.i++
191   @for d in $(ran) ; do \
192     echo $$d ; \
193     for x in $(size) ; do \
194       $(CXX) $(CXXFLAGS) $(IFLAGS) -I$* -DNUMBER=$$x -DDATA=$$d
              Drivers/pop_driver.c++ ; \
195       ./a.out ; \
196     done \
197   done
198   @make -s -f $(thisfile) clean
199
200 n = 1048576
201 t = 4194304
202 PROFILERFLAGS = -DDEBUG -Wall -std=c++11 -pedantic -x c++ -g
203
204 $(construct_prof_tests): %.prof.construct : %/binary_heap.i++
205   @echo $* "Profiling constructor ..."
206   @cp Aids/int.i++ element.i++
207   @cp Aids/std_less.i++ comparator.i++
```

```
208   @cp Aids/$(experiment).i++ experiment.i++
209   @cp $*/binary_heap.i++ data_structure.i++
210   $(CXX) $(PROFILERFLAGS) -DNDEBUG -DPROFILING $(IFLAGS) -I$* -I/usr/
          include/valgrind/ -DMAXSIZE=$(t) -DNUMBER=$(n) -DDATA=$(ran)
          Drivers/construct_driver.c++
211   valgrind --tool=callgrind --instr-atstart=no --dump-instr=yes
          --collect-jumps=yes --callgrind-out-file=$*/construct.
          callgrind.out ./a.out
212   @make -s -f $(thisfile) clean
213
214 $(push_prof_tests): %.prof.push : %/binary_heap.i++
215   @echo $* "Profiling push ..."
216   @cp Aids/int.i++ element.i++
217   @cp Aids/std_less.i++ comparator.i++
218   @cp Aids/$(experiment).i++ experiment.i++
219   @cp $*/binary_heap.i++ data_structure.i++
220   $(CXX) $(PROFILERFLAGS) -DNDEBUG -DPROFILING $(IFLAGS) -I$* -I/usr/
          include/valgrind/ -DMAXSIZE=$(t) -DNUMBER=$(n) -DDATA=$(inc)
          Drivers/push_driver.c++
221   valgrind --tool=callgrind --instr-atstart=no --dump-instr=yes
          --collect-jumps=yes --callgrind-out-file=$*/push.callgrind.
          out ./a.out
222   @make -s -f $(thisfile) clean
223
224 $(pop_prof_tests): %.prof.pop : %/binary_heap.i++
225   @echo $* "Profiling pop ..."
226   @cp Aids/int.i++ element.i++
227   @cp Aids/std_less.i++ comparator.i++
228   @cp Aids/$(experiment).i++ experiment.i++
229   @cp $*/binary_heap.i++ data_structure.i++
230   $(CXX) $(PROFILERFLAGS) -DNDEBUG -DPROFILING $(IFLAGS) -I$* -I/usr/
          include/valgrind/ -DMAXSIZE=$(t) -DNUMBER=$(n) -DDATA=$(ran)
          Drivers/pop_driver.c++
231   valgrind --tool=callgrind --instr-atstart=no --dump-instr=yes
          --collect-jumps=yes --callgrind-out-file=$*/pop.callgrind.out
           ./a.out
232   @make -s -f $(thisfile) clean
233
234 clean:
235   @rm -rf *~ a.out element.i++ comparator.i++ experiment.i++ data
          _structure.i++ temp 2>/dev/null
```