Conceptual Frameworks for Constructing Iterators for Compound Data Structures Electronic Appendix I: Component-Iterator and Rank-Iterator Classes

Jyrki Katajainen and Andreas Milton Maniotis

Department of Computer Science
University of Copenhagen, 2100 Copenhagen East, Denmark
{jyrki | maniotis}@diku.dk

Abstract. This report is an electronic appendix to the paper "Conceptual Frameworks for Constructing Iterators for Compound Data Structures" that has been submitted for publication. This report describes the proof-of-concept prototypes for

Component iterator: It can be used to provide iterators for components needing a bidirectional iterator, provided that the components support the following seven getter functions for navigation:

- `owner_type* up() const;`
- `self_type* next() const;`
- `self_type* previous() const;`
- `subordinate_type* down() const;`
- `bool is_end() const;`
- `iterator begin() const;`
- `iterator end() const;`

The components at the bottom layer should also support the function

- `value_type const& element() const;`

Moreover, we need 2-5 setter functions to update the information. The actual number depends on which changes are allowed to be done by others.

Rank iterator: It can be used to provide iterators for components needing a random-access iterator; the only requirement is that the components retain indices valid, i.e. the ith element must have the same index its entire lifetime.

This report also gives the other programs described in the use cases and used in the experiments of this paper. All programs have been revised, corrected, and cleaned up after the submission.

Keywords. Compound data structures, iterators, template metaprogramming, static collections, mergeable collections, space-efficient vectors, efficiency, exception safety, configurability

CPH STL Report 2012-3, January 2013.

Release date

2013-01-11

Included files

Collection of collections

§ 1 *factory.h++*

```
/*
  This factory can create any kind of objects.

  Author: Jyrki Katajainen © 2012
*/

#include <utility> // std::forward

namespace cphstl {

  template <typename A>
  class factory {
  public:

    typedef A allocator_type;
    typedef typename A::value_type object_type;

  protected:

    A allocator;

  public:

    explicit factory(A const & a = A())
      : allocator(a) {
    }

    factory(factory const & other)
      : allocator(other.allocator) {
    }

    factory & operator=(factory const & other) {
      allocator = other.allocator;
      return *this;
    }

    template <typename... Args>
    object_type* create(Args&&... args) {
      object_type* p = allocator.allocate(1);
      try {
        new (p) object_type(std::forward<Args>(args)...);
      }
      catch (...) {
        destroy(p);
        throw;
      }
      return p;
    }

    void destroy(object_type* p) {
      p->~object_type();
      allocator.deallocate(p, 1);
      p = nullptr;
    }
  };
}
```

§ 2 *collection-use-case.m++*

```
/* -*- C++ -*-
   This module shows how to use a collection of collections.

   Author: Jyrki Katajainen © 2012, 2013
*/

#ifndef __CPHSTL_COLLECTION_USE_CASE__
#define __CPHSTL_COLLECTION_USE_CASE__

#include <cassert>
#include "component_iterator.h++"
#include <cstddef> // std::size_t
#include "factory.h++"
#include <iostream>
#include <memory>
#include <type_traits> // std::conditional, std::is_same

namespace cphstl {

  // metaprogramming aids

  template <typename T>
  struct depth {
    static std::size_t const value = 0;
  };

  template <template <typename, typename...> class C, typename F, typename...
      Rest>
  struct depth<C<F, Rest...> > {
    static std::size_t const value = depth<F>::value + 1;
  };

  template <int>
  class null {
  public:

    typedef null owner_type;
  };

  // forward declarations

  template <typename V, typename A, typename M>
  class node;

  template <typename S, typename A, typename M>
  class collection;

  // class definitions

  template <typename V, typename A, typename M = cphstl::null<0> >
  class node {
  public:

    typedef V value_type;
    typedef node node_type;
    typedef typename A::template rebind<V>::other allocator_type;
    typedef M mediator_type;
    typedef typename std::conditional<std::is_same<M, cphstl::null<0> >::value,
        node, typename M::owner_type>::type owner_type;
    typedef cphstl::component_iterator<1, node_type> iterator;
    typedef cphstl::component_iterator<1, node_type const> const_iterator;

  protected:

    node* predecessor;
```

```
64      node* successor;
65      bool is_sentinel;
66      union {
67        V value;
68        owner_type* owner;
69      };
70
71    private:
72
73      node(node const &);
74      node & operator=(node const &);
75
76    public:
77
78      node();
79      node(owner_type*);
80      node(V const & v);
81      ~node();
82
83      owner_type* up() const;
84      node* previous() const;
85      node* next() const;
86      V const & element() const;
87      bool is_end() const;
88      iterator begin() const;
89      iterator end() const;
90
91      void previous(node*);
92      void next(node*);
93      V & element();
94    };
95
96    template <typename S, typename A, typename M = cphstl::null<0> >
97    class collection {
98    public:
99
100     typedef typename S::value_type value_type;
101     typedef typename S::node_type node_type;
102     typedef S subordinate_type;
103     typedef typename A::template rebind<S>::other allocator_type;
104     typedef M mediator_type;
105     typedef typename std::conditional<std::is_same<M, cphstl::null<0> >::value,
          collection, typename M::owner_type>::type owner_type;
106     typedef std::size_t size_type;
107     typedef collection self_type;
108     typedef cphstl::component_iterator<cphstl::depth<self_type>::value, self_type
          > iterator;
109     typedef cphstl::component_iterator<cphstl::depth<self_type>::value, self_type
          const> const_iterator;
110
111 protected:
112
113     owner_type* owner;
114     collection* successor;
115     collection* predecessor;
116     S* head;
117     bool is_sentinel;
118     size_type cardinality;
119     cphstl::factory<allocator_type> sentinel_factory;
120
121   public:
122
123     collection();
124     collection(owner_type*);
125     ~collection();
```

```
126
127     size_type size() const;
128     owner_type* up() const;
129     collection* next() const;
130     collection* previous() const;
131     subordinate_type* down() const;
132     bool is_end() const;
133     iterator begin() const;
134     iterator end() const;
135
136     void next(collection*);
137     void previous(collection*);
138     void insert(subordinate_type*);
139     void extract(subordinate_type*);
140     void merge(collection*);
141
142     template <typename F>
143     void clear(F &);
144   };
145 }
146
147 // implementation details
148
149 #include "collection.i++"
150 #include "node.i++"
151
152 #if defined(UNITTEST_COLLECTION_USE_CASE)
153
154 #include <iostream>
155 #include <iterator>
156 #include <memory>
157
158 template <typename C>
159 void show(C const & collection) {
160   std::cout << "[";
161   for (auto j = collection.begin(); j != collection.end(); ++j) {
162     if (j != collection.begin()) {
163       std::cout << " ";
164     }
165     std::cout << *j;
166   }
167   std::cout << "]" << std::endl;
168 }
169
170 template <typename C>
171 void show_reverse(C const & collection) {
172   typedef typename C::iterator our_iterator;
173   std::reverse_iterator<our_iterator> reverse_first(collection.end());
174   std::reverse_iterator<our_iterator> reverse_end(collection.begin());
175   std::cout << "[";
176   for (auto j = reverse_first; j != reverse_end; ++j) {
177     if (j != reverse_first) {
178       std::cout << " ";
179     }
180     std::cout << *j;
181   }
182   std::cout << "]" << std::endl;
183 }
184
185 template <typename A, typename X>
186 void collection_test() {
187   typedef X collection_type;
188   typedef typename X::value_type V;
189   typedef typename X::node_type N;
190   typedef typename A::template rebind<N>::other B;
```

```
191   typedef cphstl::factory<B> F;
192
193   B node_allocator;
194   F node_factory = F(node_allocator);
195   X collection;
196
197   for (int i = 0; i != 4; ++i) {
198     N* p = node_factory.create(V(i));
199     collection.insert(p);
200   }
201
202   std::cout << "x: ";
203   show(collection);
204
205   collection.clear(node_factory);
206 }
207
208 template <typename A, typename Y>
209 void collection_of_collections_test() {
210   typedef typename Y::value_type V;
211   typedef typename Y::node_type L1;
212   typedef typename A::template rebind<L1>::other A1;
213   typedef typename L1::owner_type L2;
214   typedef typename A::template rebind<L2>::other A2;
215   typedef Y L3;
216
217   typedef cphstl::factory<A1> F1;
218   typedef cphstl::factory<A2> F2;
219
220   A1 node_allocator;
221   F1 node_factory = F1(node_allocator);
222   A2 collection_allocator;
223   F2 collection_factory = F2(collection_allocator);
224   L3 top_collection;
225
226   L2* a = collection_factory.create();
227   for (int i = 0; i != 4; ++i) {
228     L1* p = node_factory.create(V(i));
229     (*a).insert(p);
230   }
231   top_collection.insert(a);
232   std::cout << "a: ";
233   show(*a);
234   std::cout << "reverse a: ";
235   show_reverse(*a);
236
237   L2* b = collection_factory.create();
238   for (int i = 4; i != 7; ++i) {
239     L1* p = node_factory.create(V(i));
240     (*b).insert(p);
241   }
242   top_collection.insert(b);
243   std::cout << "b: ";
244   show(*b);
245   std::cout << "reverse b: ";
246   show_reverse(*b);
247
248   std::cout << "a and b: ";
249   show(top_collection);
250   std::cout << "reverse a and b: ";
251   show_reverse(top_collection);
252
253   (*a).merge(b);
254   top_collection.extract(b);
255   collection_factory.destroy(b);
```

```
256
257   std::cout << "a union b: ";
258   show(*a);
259   std::cout << "reverse a union b: ";
260   show_reverse(*a);
261
262   top_collection.extract(a);
263   (*a).clear(node_factory);
264   collection_factory.destroy(a);
265 }
266
267 template <
268   typename V,
269   typename A,
270   template <typename, typename, typename> class N,
271   template <typename, typename, typename> class C
272 >
273 class simple_entangle {
274 public:
275
276   typedef C<N<V, A, simple_entangle<V, A, N, C> >, A, cphstl::null<0> >
          owner_type;
277 };
278
279 template <typename V, typename A>
280 using bottom_layer
281 = cphstl::node<V, A, simple_entangle<V, A, cphstl::node, cphstl::collection> >;
282
283 template <
284   int n,
285   typename V,
286   typename A,
287   template <typename...>  class N,
288   template <typename...>  class C,
289   template <typename...> class T
290 >
291 class entangle;
292
293 template <
294   typename V,
295   typename A,
296   template <typename, typename, typename>  class N,
297   template <typename, typename, typename>  class C,
298   template <typename...> class T
299 >
300 class entangle<0, V, A, N, C, T> {
301 public:
302
303   typedef C<N<V, A, entangle<0, V, A, N, C, T> >, A, entangle<1, V, A, N, C, T> >
          owner_type;
304 };
305
306 template <
307   typename V,
308   typename A,
309   template <typename, typename, typename>  class N,
310   template <typename, typename, typename>  class C,
311   template <typename...> class T
312 >
313 class entangle<1, V, A, N, C, T> {
314 public:
315
316   typedef  T<C<N<V, A, entangle<0, V, A, N, C, T> >, A, entangle<1, V, A, N, C, T
          > >, A, cphstl::null<0> > owner_type;
317 };
```

```
318
319 template <typename V, typename A>
320 using multiple_collections
321 = cphstl::collection<cphstl::collection<cphstl::node<V, A, entangle<0, V, A,
        cphstl::node, cphstl::collection, cphstl::collection> >, A, entangle<1, V, A,
         cphstl::node, cphstl::collection, cphstl::collection> >, A, cphstl::null<0>
        >;
322
323 int main(int, char**) {
324   typedef int V;
325   typedef std::allocator<V> A;
326   typedef bottom_layer<V, A> N;
327   typedef cphstl::collection<N, A> X;
328
329   collection_test<A, X>();
330
331   typedef multiple_collections<V, A> Y;
332   collection_of_collections_test<A, Y>();
333
334   std::cout << "Tests passed" << std::endl;
335   return 0;
336 }
337
338 #endif
339 #endif
```

§ 3 *node.i++*

```
 1 namespace cphstl {
 2
 3   // default constructor
 4
 5   template <typename V, typename A, typename M>
 6   node<V, A, M>::node()
 7     : predecessor(this), successor(this), is_sentinel(true) {
 8     owner = nullptr;
 9   }
10
11   // parametrized constructors
12
13   template <typename V, typename A, typename M>
14   node<V, A, M>::node(typename node<V, A, M>::owner_type* o)
15     : predecessor(this), successor(this), is_sentinel(true) {
16     owner = o;
17   }
18
19   template <typename V, typename A, typename M>
20   node<V, A, M>::node(V const & v)
21     : predecessor(this), successor(this), is_sentinel(false), value(v) {
22   }
23
24   // destructor
25
26   template <typename V, typename A, typename M>
27   node<V, A, M>::~node() {
28   }
29
30   // previous
31
32   template <typename V, typename A, typename M>
33   node<V, A, M>* node<V, A, M>::previous() const {
34     return predecessor;
35   }
36
```

```
37   template <typename V, typename A, typename M>
38   void node<V, A, M>::previous(node<V, A, M>* p) {
39     predecessor = p;
40   }
41
42   // next
43
44   template <typename V, typename A, typename M>
45   node<V, A, M>* node<V, A, M>::next() const {
46     return successor;
47   }
48
49   template <typename V, typename A, typename M>
50   void node<V, A, M>::next(node<V, A, M>* s) {
51     successor = s;
52   }
53
54   // element
55
56   template <typename V, typename A, typename M>
57   V const & node<V, A, M>::element() const {
58     return value;
59   }
60
61   template <typename V, typename A, typename M>
62   V & node<V, A, M>::element() {
63     return value;
64   }
65
66   // iterator support
67
68   template <typename V, typename A, typename M>
69   bool node<V, A, M>::is_end() const {
70     return is_sentinel;
71   }
72
73   template <typename V, typename A, typename M>
74   typename node<V, A, M>::iterator node<V, A, M>::begin() const {
75     return (iterator)(node*) this;
76   }
77
78   template <typename V, typename A, typename M>
79   typename node<V, A, M>::iterator node<V, A, M>::end() const {
80     return (iterator)(node*) this;
81   }
82
83   template <typename V, typename A, typename M>
84   typename node<V, A, M>::owner_type* node<V, A, M>::up() const {
85     node const* t = this;
86     while (! (*t).is_end()) {
87       t = (*t).next();
88     }
89     return (*t).owner;
90   }
91 }
```

§ 4 *collection.i++*

```
1 /*
2    Implementing cphstl::collection
3
4    Author: Jyrki Katajainen © 2012
5 */
6
```

```
7  namespace cphstl {
8
9    template <typename S, typename A, typename M>
10   collection<S, A, M>::collection()
11     : owner(nullptr), successor(this), predecessor(this), is_sentinel(true),
12       cardinality(0) {
13     sentinel_factory = cphstl::factory<allocator_type>(allocator_type());
14     head = sentinel_factory.create((collection*) this);
15     (*head).previous(head);
16     (*head).next(head);
17   }
18
19   template <typename S, typename A, typename M>
20   collection<S, A, M>::collection(typename collection<S, A, M>::owner_type* o)
21     : owner(o), successor(this), predecessor(this), is_sentinel(true),
22       cardinality(0) {
23     sentinel_factory = cphstl::factory<allocator_type>(allocator_type());
24     head = sentinel_factory.create((collection*) this);
25     (*head).previous(head);
26     (*head).next(head);
27   }
28
29   template <typename S, typename A, typename M>
30   collection<S, A, M>::~collection() {
31     sentinel_factory.destroy(head);
32     head = nullptr;
33   }
34
35   template <typename S, typename A, typename M>
36   typename collection<S, A, M>::owner_type* collection<S, A, M>::up() const {
37     return owner;
38   }
39
40   template <typename S, typename A, typename M>
41   collection<S, A, M>* collection<S, A, M>::next() const {
42     return successor;
43   }
44
45   template <typename S, typename A, typename M>
46   collection<S, A, M>* collection<S, A, M>::previous() const {
47     return predecessor;
48   }
49
50   template <typename S, typename A, typename M>
51   S* collection<S, A, M>::down() const {
52     return head;
53   }
54
55   template <typename S, typename A, typename M>
56   bool collection<S, A, M>::is_end() const {
57     return is_sentinel;
58   }
59
60   template <typename S, typename A, typename M>
61   typename collection<S, A, M>::size_type collection<S, A, M>::size() const {
62     return cardinality;
63   }
64
65   template <typename S, typename A, typename M>
66   typename collection<S, A, M>::iterator collection<S, A, M>::begin() const {
67     return down() → next() → begin();
68   }
69
70   template <typename S, typename A, typename M>
71   typename collection<S, A, M>::iterator collection<S, A, M>::end() const {
```

```
72      return down() → end();
73    }
74
75    template <typename S, typename A, typename M>
76    void collection<S, A, M>::next(collection* v) {
77      successor = v;
78    }
79
80    template <typename S, typename A, typename M>
81    void collection<S, A, M>::previous(collection* v) {
82      predecessor = v;
83    }
84
85    template <typename S, typename A, typename M>
86    void collection<S, A, M>::insert(S* q) {
87      S* tail = (*head).previous();
88      (*tail).next(q);
89      (*q).previous(tail);
90      (*q).next(head);
91      (*head).previous(q);
92      ++cardinality;
93    }
94
95    template <typename S, typename A, typename M>
96    void collection<S, A, M>::extract(S* q) {
97      S* p = (*q).previous();
98      S* r = (*q).next();
99      (*p).next(r);
100     (*r).previous(p);
101     (*q).next(nullptr);
102     (*q).previous(nullptr);
103     --cardinality;
104   }
105
106   template <typename S, typename A, typename M>
107   void collection<S, A, M>::merge(collection* other) {
108     if ((*other).size() == 0) {
109       return;
110     }
111     S* tail = (*head).previous();
112     S* first = (*other).head → next();
113     S* last = (*other).head → previous();
114     (*tail).next(first);
115     (*first).previous(tail);
116     (*last).next(head);
117     (*head).previous(last);
118     cardinality += (*other).cardinality;
119     (*other).cardinality = 0;
120     (*other).head → previous((*other).head);
121     (*other).head → next((*other).head);
122   }
123
124   template <typename S, typename A, typename M>
125   template <typename F>
126   void collection<S, A, M>::clear(F & factory) {
127     size_type n = size();
128     for (size_type i = 0; i != n; ++i) {
129       S* p = down();
130       p = (*p).next();
131       extract(p);
132       factory.destroy(p);
133     }
134     cardinality = 0;
135   }
136 }
```

Component iterator

§ 5 *component-iterator.h++*

```
1  /*
2    A component iterator encapsulates a node at the bottom layer of a
3    component hierarchy.
4
5    The components should provide the following functions for navigation:
6    is_end: true if the current location is one past the end at this layer
7    next: the successor of the current location at this layer
8    previous: the predecessor of the current location at this layer.
9    up: get to the owner of the current location at the layer above
10   down: get to the first subordinate of the current location at the layer below
11   begin: go to the first location at the bottom layer
12   end: go to the one-past-the-end location at the bottom layer
13   element: returns the element encapsulated at the bottom layer
14
15   Authors: Jyrki Katajainen, Bo Simonsen © 2006, 2008, 2012
16 */
17
18 #include <cstddef> // std::ptrdiff_t, std::nullptr_t
19 #include <iterator> // std::bidirectional_iterator_tag
20 #include <type_traits> // std::conditional, std::is_const, std::remove_const
21
22 namespace cphstl {
23
24   // Forward declaration
25
26   template <int, typename>
27   class component_iterator;
28
29   // At the bottom layer, for nodes no iteration is provided
30
31   class trivial_iterator_tag {
32   };
33
34   template <typename N>
35   class component_iterator<1, N> {
36   public:
37
38     typedef N node_type;
39     typedef typename std::conditional<std::is_const<N>::value, typename std::
           remove_const<N>::type, N const>::type opposite;
40     typedef typename N::value_type value_type;
41     typedef std::size_t size_type;
42     typedef std::ptrdiff_t difference_type;
43    typedef typename std::conditional<std::is_const<N>::value, value_type const*,
         value_type*>::type pointer;
44     typedef typename std::conditional<std::is_const<N>::value, value_type const &
          , value_type &>::type reference;
45     typedef cphstl::trivial_iterator_tag iterator_category;
46
47     friend class component_iterator<1, opposite>;
48
49     template <int n, typename C>
50     friend class component_iterator;
51
52   protected:
53
54     N* link;
55
56   public:
57
```

```
58      component_iterator(N* p)
59        : link(p) {
60      }
61
62      operator N*() const {
63        return link;
64      }
65
66      component_iterator()
67        : link(nullptr) {
68      }
69
70      component_iterator(component_iterator const & other)
71        : link(other.link) {
72      }
73
74      component_iterator & operator=(component_iterator const & other) {
75        link = other.link;
76        return *this;
77      }
78
79      ~component_iterator() {
80      }
81
82      reference operator*() const {
83        return (*link).element();
84      }
85
86      pointer operator->() const {
87        return &(*link).element();
88      }
89
90      template <int n, typename C>
91      bool operator==(component_iterator<n, C> const & other) const {
92        return link == other.link;
93      }
94
95      template <int n, typename C>
96      bool operator!=(component_iterator<n, C> const & other) const {
97        return link != other.link;
98      }
99    };
100
101   // Two layers: a collection consisting of nodes
102
103   template <typename C>
104   class component_iterator<2, C>
105     : public component_iterator<1, typename C::node_type> {
106
107   public:
108
109     typedef C collection_type;
110     typedef typename C::node_type N;
111     typedef typename std::conditional<std::is_const<C>::value, typename std::
            remove_const<C>::type, C const>::type opposite;
112     typedef component_iterator<1, N> base_type;
113     typedef std::bidirectional_iterator_tag iterator_category;
114
115     // friends
116
117     friend class component_iterator<2, opposite>;
118
119   public:
120
121     component_iterator(component_iterator<1, N> const & a)
```

```
122       : base_type(a.link) {
123     }
124
125     component_iterator(component_iterator<2, C const> const & a)
126       : base_type(a.link) {
127     }
128
129     component_iterator & operator++() {
130       N* q = (*this).link;
131       q = (*q).next();
132       (*this).link = q;
133       return *this;
134     }
135
136     component_iterator & operator--() {
137       N* q = (*this).link;
138       q = (*q).previous();
139       (*this).link = q;
140       return *this;
141     }
142   };
143
144   // Three layers: a collection of collections of nodes
145
146   template <typename C>
147   class component_iterator<3, C>
148     : public component_iterator<1, typename C::node_type> {
149
150   public:
151
152     typedef C collection_type;
153     typedef typename C::node_type N;
154     typedef typename std::conditional<std::is_const<C>::value, typename std::
          remove_const<C>::type, C const>::type opposite;
155     typedef component_iterator<1, N> base_type;
156     typedef std::bidirectional_iterator_tag iterator_category;
157
158     // friends
159
160     friend class component_iterator<3, opposite>;
161
162   public:
163
164     component_iterator(component_iterator<2, typename C::subordinate_type> const
          & a)
165       : base_type(a.link) {
166     }
167
168     component_iterator(component_iterator<3, C const> const & a)
169       : base_type(a.link) {
170     }
171
172     component_iterator & operator++() {
173       N* r = (*this).link;
174       r = (*r).next();
175       if ((*r).is_end()) {
176         auto q = (*r).up();
177         r = (*q).next() -> begin();
178       }
179       (*this).link = r;
180       return *this;
181     }
182
183     component_iterator & operator--() {
184       N* r = (*this).link;
```

```
185       r = (*r).previous();
186       if ((*r).is_end()) {
187         auto q = (*r).up();
188         r = (*q).previous() → end();
189         r = (*r).previous();
190       }
191       (*this).link = r;
192       return *this;
193     }
194   };
195 }
```

Vectors

§ 6 *dynamic-array.h++*

```
1  /*
2    This is an experimental array which is based on doubling and halving.
3
4    Author: Jyrki Katajainen @ 2012
5  */
6
7  #include <cstddef> // std::size_t and std::ptrdiff_t
8  #include <memory> // std::allocator and std::uninitialized_copy
9  #include "rank-iterator.h++"
10
11 namespace cphstl {
12
13   template <typename V, typename A = std::allocator<V> >
14   class dynamic_array {
15   public:
16
17     typedef V value_type;
18     typedef typename A::template rebind<V>::other allocator_type;
19     typedef V & reference;
20     typedef V const & const_reference;
21     typedef V* pointer;
22     typedef V const* const_pointer;
23     typedef std::ptrdiff_t difference_type;
24     typedef std::size_t size_type;
25     typedef dynamic_array<V, allocator_type> self_type;
26     typedef cphstl::rank_iterator<self_type> iterator;
27     typedef cphstl::rank_iterator<self_type const> const_iterator;
28
29   protected:
30
31     V* X;
32     size_type X_size;
33     size_type X_capacity;
34     allocator_type allocator;
35
36     V* address(size_type rank) const {
37       return X + rank;
38     }
39
40   public:
41
42     explicit dynamic_array(A const & a = A())
43       : X(nullptr), X_size(0), X_capacity(1), allocator(a) {
44       X = allocator.allocate(2);
45     }
46
47     ~dynamic_array() {
48       clear();
```

```
49      }
50
51      void swap(dynamic_array & other) {
52        std::swap(X, other.X);
53        std::swap(X_size, other.X_size);
54        std::swap(X_capacity, other.X_capacity);
55        std::swap(allocator, other.allocator);
56      }
57
58      const_iterator begin() const {
59        return const_iterator(std::make_pair(this, size_type(0)));
60      }
61
62      iterator begin() {
63        return iterator(std::make_pair(this, size_type(0)));
64      }
65
66      const_iterator end() const {
67        return const_iterator(std::make_pair(this, size_type(X_size)));
68      }
69
70      iterator end() {
71        return iterator(std::make_pair(this, size_type(X_size)));
72      }
73
74      allocator_type get_allocator() const {
75        return allocator;
76      }
77
78      size_type size() const {
79        return X_size;
80      }
81
82      size_type capacity() const {
83        return X_capacity;
84      }
85
86      const_reference operator[](size_type i) const {
87        return *address(i);
88      }
89
90      reference operator[](size_type i) {
91        return *address(i);
92      }
93
94      void push_back(V const & x) {
95        if (X_size == X_capacity) {
96          V* tmp = allocator.allocate(2 * X_size);
97          std::uninitialized_copy(X, X + X_size, tmp);
98          allocator.deallocate(X, X_capacity);
99          std::swap(X, tmp);
100         X_capacity = 2 * X_size;
101       }
102       allocator.construct(X + X_size, x);
103       ++X_size;
104     }
105
106     void pop_back() {
107       if (4 * X_size == X_capacity) {
108         V* tmp = allocator.allocate(2 * X_size);
109         std::uninitialized_copy(X, X + X_size, tmp);
110         allocator.deallocate(X, X_capacity);
111         std::swap(X, tmp);
112         X_capacity = 2 * X_size;
113       }
```

```
114       --X_size;
115       allocator.destroy(X + X_size);
116     }
117
118     void clear() {
119       for (size_type i = 0; i != X_size; ++i) {
120         allocator.destroy(X + i);
121       }
122       allocator.deallocate(X, X_capacity);
123       X = nullptr;
124       X_size = 0;
125       X_capacity = 0;
126     }
127   };
128 }
```

§ 7 *folklore.h++*

```
1  /*
2    This is an experimental implementation of the dynamic array based on
3    doubling, halving, and inceremental copying.
4
5    Author: Jyrki Katajainen @ 2012
6  */
7
8  #include <algorithm> // std::max
9  #include <cstddef> // std::size_t and std::ptrdiff_t
10 #include <iterator>
11 #include <memory> // std::allocator
12 #include "rank-iterator.h++"
13
14 namespace cphstl {
15
16   template <typename V, typename A = std::allocator<V> >
17   class folklore {
18   public:
19
20     typedef V value_type;
21     typedef typename A::template rebind<V>::other allocator_type;
22     typedef V & reference;
23     typedef V const & const_reference;
24     typedef V* pointer;
25     typedef V const* const_pointer;
26     typedef std::ptrdiff_t difference_type;
27     typedef std::size_t size_type;
28     typedef folklore<V, allocator_type> self_type;
29     typedef cphstl::rank_iterator<self_type> iterator;
30     typedef cphstl::rank_iterator<self_type const> const_iterator;
31
32   protected:
33
34     V* X;
35     V* Y;
36     size_type X_size;
37     size_type Y_size;
38     size_type X_capacity;
39     size_type Y_capacity;
40     allocator_type allocator;
41
42     V* address(size_type rank) const {
43       if (rank < X_size) {
44         return X + rank;
45       }
46       return Y + rank;
```

```
47      }
48
49    public:
50
51      explicit folklore(A const & a = A())
52        : Y(nullptr), X_size(0), Y_size(0), X_capacity(1), Y_capacity(0),
53          allocator(a) {
54        X = allocator.allocate(1);
55      }
56
57      ~folklore() {
58        clear();
59      }
60
61      const_iterator begin() const {
62        return const_iterator(std::make_pair(this, size_type(0)));
63      }
64
65      iterator begin() {
66        return iterator(std::make_pair(this, size_type(0)));
67      }
68
69      const_iterator end() const {
70        size_type max = std::max(X_size, Y_size);
71        return const_iterator(std::make_pair(this, max));
72      }
73
74      iterator end() {
75        size_type max = std::max(X_size, Y_size);
76        return iterator(std::make_pair(this, max));
77      }
78
79      allocator_type get_allocator() const {
80        return allocator;
81      }
82
83      size_type size() const {
84        return std::max(X_size, Y_size);
85      }
86
87      size_type capacity() const {
88        if (Y_size != 0) {
89          return Y_capacity;
90        }
91        return X_capacity;
92      }
93
94      const_reference operator[](size_type i) const {
95        return *address(i);
96      }
97
98      reference operator[](size_type i) {
99        return *address(i);
100     }
101
102     void push_back(V const & x) {
103       if (Y_size == 0 && X_size < X_capacity) {
104         allocator.construct(X + X_size, x);
105         ++X_size;
106         return;
107       }
108       if (Y_size == 0 && X_size == X_capacity) {
109         Y_capacity = 2 * X_size;
110         Y = allocator.allocate(Y_capacity);
111         Y_size = X_size;
```

```
112       }
113       --X_size;
114       allocator.construct(Y + X_size, std::move(X[X_size]));
115       allocator.destroy(X + X_size);
116       allocator.construct(Y + Y_size, x);
117       ++Y_size;
118       if (X_size == 0) {
119         allocator.deallocate(X, X_capacity);
120         X = Y;
121         X_size = Y_size;
122         X_capacity = Y_capacity;
123         Y = nullptr;
124         Y_size = 0;
125         Y_capacity = 0;
126       }
127     }
128
129     void pop_back() {
130       if (Y_size == 0 && 4 * X_size > X_capacity) {
131         --X_size;
132         allocator.destroy(X + X_size);
133         return;
134       }
135       if (Y_size == 0 && 4 * X_size == X_capacity) {
136         Y_capacity = 2 * X_size;
137         Y = allocator.allocate(Y_capacity);
138         Y_size = X_size;
139       }
140       --X_size;
141       allocator.construct(Y + X_size, std::move(X[X_size]));
142       allocator.destroy(X + X_size);
143       if (X_size != 0) {
144         --X_size;
145         allocator.construct(Y + X_size, std::move(X[X_size]));
146         allocator.destroy(X + X_size);
147       }
148       --Y_size;
149       allocator.destroy(Y + Y_size);
150       if (X_size == 0) {
151         allocator.deallocate(X, X_capacity);
152         X = Y;
153         X_size = Y_size;
154         X_capacity = Y_capacity;
155         Y = nullptr;
156         Y_size = 0;
157         Y_capacity = 0;
158       }
159     }
160
161     void clear() {
162       for (size_type i = 0; i != X_size; ++i) {
163         allocator.destroy(X + i);
164       }
165       for (size_type j = 0; j != Y_size; ++j) {
166         allocator.destroy(Y + j);
167       }
168       allocator.deallocate(X, X_capacity);
169       allocator.deallocate(Y, Y_capacity);
170       X_size = 0;
171       Y_size = 0;
172       X_capacity = 0;
173       Y_capacity = 0;
174     }
175   };
176 }
```

§ 8 *pile-of-segments.h++*

```
/*
  This is an experimental implementation of a levelwise-allocated
  pile; each level stores a fixed-capacity segment and the header is a
  (worst-case efficient) vector.

  Author: Jyrki Katajainen @ 2012
*/

#include <algorithm> // std::max
#include <climits> // CHAR_BIT
#include <cmath> // std::ilogb
#include <cstddef> // std::size_t and std::ptrdiff_t
#include "folklore.h++"
#include <memory> // std::allocator
#include "rank-iterator.h++"
#include <vector>

std::size_t const w_minus_one = sizeof(std::size_t) * CHAR_BIT - 1;

unsigned int population_count(unsigned int j) {
  return __builtin_popcount(j);
}

unsigned long population_count(unsigned long j) {
  return __builtin_popcountl(j);
}

unsigned long long population_count(unsigned long long j) {
  return __builtin_popcountll(j);
}

unsigned int whole_number_logarithm(unsigned int j) {
  return (unsigned int) std::ilogb(j);
  // return (w_minus_one - __builtin_clz(j));
}

unsigned long whole_number_logarithm(unsigned long j) {
  return (unsigned long) std::ilogb(j);
  // // if ((w_minus_one - __builtin_clzl(j)) != std::ilogb(j)) {
  // }
  // return w_minus_one - __builtin_clzl(j);
}

unsigned long long whole_number_logarithm(unsigned long long j) {
  return (unsigned long long) std::ilogb(j);
  // return (w_minus_one - __builtin_clzll(j));
}

namespace cphstl {

  template <typename V, typename A = std::allocator<V> >
  class pile_of_segments {
  public:

    typedef V value_type;
    typedef typename A::template rebind<V>::other allocator_type;
    typedef V & reference;
    typedef V const & const_reference;
    typedef V* pointer;
    typedef V const* const_pointer;
    typedef std::ptrdiff_t difference_type;
    typedef std::size_t size_type;
    typedef pile_of_segments<V, allocator_type> self_type;
```

```
64    typedef cphstl::rank_iterator<self_type> iterator;
65    typedef cphstl::rank_iterator<self_type const> const_iterator;
66
67  protected:
68
69    typedef typename A::template rebind<V*>::other B;
70    typedef cphstl::folklore<V*, B> header_type;
71    // typedef std::vector<V*, B> header_type;
72    header_type header;
73    size_type n;
74    allocator_type allocator;
75
76    V* address(size_type rank) const {
77      if (rank < 2) {
78        return header[0] + rank;
79      }
80      size_type h = whole_number_logarithm(rank);
81      return header[h] + rank - (1 << h);
82    }
83
84  public:
85
86    explicit pile_of_segments(A const & a = A())
87      : header(a), n(0), allocator(a) {
88      V* p = allocator.allocate(2);
89      header.push_back(p);
90    }
91
92    ~pile_of_segments() {
93      clear();
94    }
95
96    const_iterator begin() const {
97      return const_iterator(std::make_pair(this, size_type(0)));
98    }
99
100   iterator begin() {
101     return iterator(std::make_pair(this, size_type(0)));
102   }
103
104   const_iterator end() const {
105     return const_iterator(std::make_pair(this, n));
106   }
107
108   iterator end() {
109     return iterator(std::make_pair(this, n));
110   }
111
112   allocator_type get_allocator() const {
113     return allocator;
114   }
115
116   size_type size() const {
117     return n;
118   }
119
120   size_type capacity() const {
121     return size_type(-1);
122   }
123
124   const_reference operator[](size_type i) const {
125     return *address(i);
126   }
127
128   reference operator[](size_type i) {
```

```
129       return *address(i);
130     }
131
132     void push_back(V const & x) {
133       size_type h = 1;
134       if (n > 1) {
135         h = header.size();
136         if (population_count(n) == 1) {
137           V* p = allocator.allocate(1 << h);
138           header.push_back(p);
139           h += 1;
140         }
141       }
142       allocator.construct(address(n), x);
143       n += 1;
144     }
145
146     void pop_back() {
147       n -= 1;
148       allocator.destroy(address(n));
149       size_type h = header.size();
150       if (h != 1 && population_count(n) == 1) {
151         h -= 1;
152         allocator.deallocate(header[h], 1 << h);
153         header.pop_back();
154       }
155     }
156
157     void clear() {
158       for (size_type i = 0; i != n; ++i) {
159         pop_back();
160       }
161       allocator.deallocate(header[0], 2);
162       header.pop_back();
163     }
164   };
165 }
```

§ 9 *hashed-array-tree.h++*

```
1  /*
2    This is an experimental implementation of a hashed-array tree for
3    which capacity will be fixed at construction time.
4
5    Author: Jyrki Katajainen @ 2012
6  */
7
8  #include <cmath> // std::ilogb
9  #include <cstddef> // std::size_t and std::ptrdiff_t
10 #include "dynamic-array.h++"
11 #include "rank-iterator.h++"
12 #include <vector>
13
14 namespace cphstl {
15
16   template <typename V, typename A = std::allocator<V> >
17   class hashed_array_tree {
18   public:
19
20     typedef V value_type;
21     typedef typename A::template rebind<V>::other allocator_type;
22     typedef V & reference;
23     typedef V const & const_reference;
24     typedef V* pointer;
```

```
25    typedef V const* const_pointer;
26    typedef std::ptrdiff_t difference_type;
27    typedef std::size_t size_type;
28    typedef hashed_array_tree<V, allocator_type> self_type;
29    typedef cphstl::rank_iterator<self_type> iterator;
30    typedef cphstl::rank_iterator<self_type const> const_iterator;
31
32  protected:
33
34    typedef typename A::template rebind<V*>::other B;
35    typedef cphstl::dynamic_array<V*, B> directory_type;
36
37    directory_type directory;
38    size_type n;
39    size_type directory_capacity;
40    size_type segment_capacity;
41    size_type mask;
42    size_type shift_amount;
43    allocator_type allocator;
44
45  public:
46
47    // Warning: not a normal vector constructor; n specifies the capacity
48
49    explicit hashed_array_tree(size_type n = 0, A const & a = A())
50      : directory(a), n(0), directory_capacity(0), segment_capacity(0),
51        mask(0), shift_amount(0), allocator(a) {
52      if (n != 0) {
53        size_type h = std::ilogb(n);
54        segment_capacity = 1 << (h / 2);
55        mask = segment_capacity - 1;
56        shift_amount = h / 2;
57        directory_capacity = (n + segment_capacity - 1) / segment_capacity;
58        // directory.reserve(directory_capacity);
59        V* p = allocator.allocate(segment_capacity);
60        directory.push_back(p);
61      }
62    }
63
64    ~hashed_array_tree() {
65      clear();
66    }
67
68    V* address(size_type rank) const {
69      return directory[rank >> shift_amount] + (rank & mask);
70    }
71
72    const_iterator begin() const {
73      return const_iterator(std::make_pair(this, size_type(0)));
74    }
75
76    iterator begin() {
77      return iterator(std::make_pair(this, size_type(0)));
78    }
79
80    const_iterator end() const {
81      return const_iterator(std::make_pair(this, n));
82    }
83
84    iterator end() {
85      return iterator(std::make_pair(this, n));
86    }
87
88    allocator_type get_allocator() const {
89      return allocator;
```

```
90      }
91
92      size_type size() const {
93        return n;
94      }
95
96      size_type capacity() const {
97        return directory_capacity * segment_capacity;
98      }
99
100     void recapacity(size_type m) {
101       clear();
102       size_type h = std::ilogb(m);
103       segment_capacity = 1 << (h / 2);
104       mask = segment_capacity - 1;
105       shift_amount = h / 2;
106       directory_capacity = (m + segment_capacity - 1) / segment_capacity;
107       V* p = allocator.allocate(segment_capacity);
108       directory.push_back(p);
109     }
110
111     const_reference operator[](size_type i) const {
112       return *address(i);
113     }
114
115     reference operator[](size_type i) {
116       return *address(i);
117     }
118
119     void push_back(V const& x) {
120       size_type delta = n & mask;
121       if (delta == 0) {
122         V* p = allocator.allocate(segment_capacity);
123         directory.push_back(p);
124       }
125       size_type h = n >> shift_amount;
126       allocator.construct(directory[h] + delta, x);
127       n += 1;
128     }
129
130     void pop_back() {
131       n -= 1;
132       V* base_address = directory[n >> shift_amount];
133       size_type delta = n & mask;
134       allocator.destroy(base_address + delta);
135       if (delta == 0 && base_address != directory[0]) {
136         allocator.deallocate(base_address, segment_capacity);
137         directory.pop_back();
138       }
139     }
140
141     void clear() {
142       for (size_type i = 0; i != n; ++i) {
143         pop_back();
144       }
145       if (directory.size() != 0) {
146         allocator.deallocate(directory[0], segment_capacity);
147         directory.pop_back();
148       }
149     }
150   };
151 }
```

§ 10 *page-table.h++*

```
/*
  This is an experimental implementation of a page-table-based vector
  often used in the implementation of the standard-library deque; each
  page is a fixed-capacity segment and the header is a (worst-case
  efficient) vector.

  Author: Jyrki Katajainen @ 2012
*/

#include <algorithm> // std::max
#include <cstddef> // std::size_t and std::ptrdiff_t
#include "folklore.h++"
#include <memory> // std::allocator
#include "rank-iterator.h++"

namespace cphstl {

  template <typename V, typename A = std::allocator<V> >
  class page_table {
  public:

    typedef V value_type;
    typedef typename A::template rebind<V>::other allocator_type;
    typedef V & reference;
    typedef V const & const_reference;
    typedef V* pointer;
    typedef V const* const_pointer;
    typedef std::ptrdiff_t difference_type;
    typedef std::size_t size_type;
    typedef page_table<V, allocator_type> self_type;
    typedef cphstl::rank_iterator<self_type> iterator;
    typedef cphstl::rank_iterator<self_type const> const_iterator;

  protected:

    size_type const page_size = 512;
    size_type const shift_amount = 9;
    size_type const mask = 511;
    typedef typename A::template rebind<V*>::other B;
    typedef cphstl::folklore<V*, B> header_type;
    header_type header;
    size_type n;
    allocator_type allocator;

    V* address(size_type rank) const {
      return header[rank >> 9] + (rank  & 511);
    }

  public:

    explicit page_table(A const & a = A())
      : header(a), n(0), allocator(a) {
      V* p = allocator.allocate(512);
      header.push_back(p);
    }

    ~page_table() {
      clear();
    }

    const_iterator begin() const {
      return const_iterator(std::make_pair(this, size_type(0)));
    }

    iterator begin() {
```

```
66        return iterator(std::make_pair(this, size_type(0)));
67      }
68
69      const_iterator end() const {
70        return const_iterator(std::make_pair(this, n));
71      }
72
73      iterator end() {
74        return iterator(std::make_pair(this, n));
75      }
76
77      allocator_type get_allocator() const {
78        return allocator;
79      }
80
81      size_type size() const {
82        return n;
83      }
84
85      size_type capacity() const {
86        return size_type(-1);
87      }
88
89      const_reference operator[](size_type i) const {
90        return *address(i);
91      }
92
93      reference operator[](size_type i) {
94        return *address(i);
95      }
96
97      void push_back(V const& x) {
98        size_type delta = n & 511;
99        if (delta == 0) {
100         V* p = allocator.allocate(512);
101         header.push_back(p);
102       }
103       size_type h = n >> 9;
104       allocator.construct(header[h] + delta, x);
105       n += 1;
106     }
107
108     void pop_back() {
109       n -= 1;
110       V* base_address = header[n >> 9];
111       size_type delta = n & 511;
112       allocator.destroy(base_address + delta);
113       if (delta == 0 && base_address != header[0]) {
114         allocator.deallocate(base_address, 512);
115         header.pop_back();
116       }
117     }
118
119     void clear() {
120       for (size_type i = 0; i != n; ++i) {
121         pop_back();
122       }
123       allocator.deallocate(header[0], 512);
124       header.pop_back();
125     }
126   };
127 }
```

Rank iterator

§ 11 *rank-iterator.h++*

```
/*
  A rank iterator is a (pointer, rank) pair where the pointer refers
  to a data structure, called a realizator in our papers, that
  contains the underlying location and the rank is the index of that
  location within the data structure.

  Authors: Jyrki Katajainen, Bo Simonsen © 2008, 2012
*/

#include <cstddef> // std::size_t and std::ptrdiff_t
#include <iterator> // std::random_access_iterator_tag
#include <type_traits> // std::conditional, std::is_const, std::remove_const
#include <utility> // std::pair

namespace cphstl {

  template <typename R>
  class rank_iterator {

  public:

    // types

    typedef R realizator_type;
    typedef typename std::conditional<std::is_const<R>::value, typename std::
        remove_const<R>::type, R const>::type opposite;
    typedef std::ptrdiff_t difference_type;
    typedef typename R::value_type value_type;
    typedef std::size_t size_type;
    typedef typename std::conditional<std::is_const<R>::value, value_type const*,
        value_type*>::type pointer;
    typedef typename std::conditional<std::is_const<R>::value, typename R::
        const_reference, typename R::reference>::type reference;
    typedef std::random_access_iterator_tag iterator_category;

    // friends

    friend R;

    friend class rank_iterator<opposite>;

    template <typename T, typename U>
    friend class std::pair;

  protected:

    realizator_type* realizator_p;
    size_type rank;

    // parameterized constructor: std::pair → iterator

    rank_iterator(std::pair<realizator_type*, size_type> const & p)
      : realizator_p(p.first), rank(p.second) {
    }

    // convertion operator: iterator → std::pair

    operator std::pair<realizator_type*, size_type>() const {
      std::pair<realizator_type*, size_type>(realizator_p, rank);
    }
```

```
58
59  public:
60
61    // default constructor
62
63    rank_iterator()
64      : realizator_p(nullptr), rank(size_type()) {
65    }
66
67    // copy constructor
68
69    rank_iterator(rank_iterator const & other)
70      : realizator_p(other.realizator_p), rank(other.rank) {
71    }
72
73    // assignment
74
75    rank_iterator & operator=(rank_iterator const & other) {
76      realizator_p = other.realizator_p;
77      rank = other.rank;
78      return *this;
79    }
80
81    // destructor
82
83    ~rank_iterator() {
84    };
85
86    // operator*
87
88    reference operator*() const {
89      return (*realizator_p)[rank];
90    }
91
92    // operator->
93
94    pointer operator->() const {
95      return &(*realizator_p)[rank];
96    }
97
98    // operator++; pre-increment
99
100   rank_iterator & operator++() {
101     ++rank;
102     return *this;
103   }
104
105   // operator++; post-increment
106
107   rank_iterator operator++(int) {
108     rank_iterator temporary = *this;
109     ++rank;
110     return temporary;
111   }
112
113   // operator--; pre-decrement
114
115   rank_iterator & operator--() {
116     --rank;
117     return *this;
118   }
119
120   // operator--; post-decrement
121
122   rank_iterator operator--(int) {
```

```
123       rank_iterator temporary = *this;
124       --rank;
125       return temporary;
126     }
127
128     // operator+=
129
130     rank_iterator & operator+=(difference_type n) {
131       rank += n;
132       return *this;
133     }
134
135     // operator-=
136
137     rank_iterator & operator-=(difference_type n) {
138       rank -= n;
139       return *this;
140     }
141
142     // operator+
143
144     rank_iterator operator+(difference_type n) const {
145       rank_iterator temporary = *this;
146       temporary.rank += n;
147       return temporary;
148     }
149
150     // operator-
151
152     rank_iterator operator-(difference_type n) const {
153       rank_iterator temporary = *this;
154       temporary.rank -= n;
155       return temporary;
156     }
157
158     // iterator distance
159
160     difference_type operator-(rank_iterator const & other) const {
161       return rank - other.rank;
162     }
163
164     // operator==
165
166     template <typename S>
167     bool operator==(rank_iterator<S> const & other) const {
168       return rank == other.rank && realizator_p == other.realizator_p;
169     }
170
171     // operator!=
172
173     template <typename S>
174     bool operator!=(rank_iterator<S> const & other) const {
175       return ! (*this == other);
176     }
177
178     // operator<
179
180     template <typename S>
181     bool operator<(rank_iterator<S> const & other) const {
182       return ((*this) - other) < 0;
183     }
184
185     // operator>
186
187     template <typename S>
```

```
188     bool operator>(rank_iterator<S> const & other) const {
189       return other < *this;
190     }
191
192     // operator<=
193
194     template <typename S>
195     bool operator<=(rank_iterator<S> const & other) const {
196       return ! (other < *this);
197     }
198
199     // operator>=
200
201     template <typename S>
202     bool operator>=(rank_iterator<S> const & other) const {
203       return ! (*this < other);
204     }
205   };
206
207   // operator+(int, iterator)
208
209   template <typename R>
210   rank_iterator<R>
211   operator+(typename R::difference_type n, rank_iterator<R> const & a) {
212     return a + n;
213   }
214 }
```

Usage

§ 12 *std.i++*

```
1 #include <vector>
2
3 typedef std::vector<E, A> X;
```

§ 13 *dynamic-array.i++*

```
1 #include "dynamic-array.h++"
2
3 typedef cphstl::dynamic_array<E, A> X;
```

§ 14 *folklore.i++*

```
1 #include "folklore.h++"
2
3 typedef cphstl::folklore<E, A> X;
```

§ 15 *pile-of-segments.i++*

```
1 #include "pile-of-segments.h++"
2
3 typedef cphstl::pile_of_segments<E, A> X;
```

§ 16 *hashed-array-tree.i++*

```
1 #include "hashed-array-tree.h++"
2
3 typedef cphstl::hashed_array_tree<E, A> X;
```

§ 17 *page-table.i++*

```
1 #include "page_table.h++"
2
3 typedef cphstl::page_table<E, A> X;
```

Drivers

§ 18 *scan-driver.c++*

```
1 #if ! defined(MAXSIZE)
2
3 #define MAXSIZE (128*1024*1024)
4
5 #endif
6
7 #include <algorithm>  // std::random_shuffle, std::make_heap, std::sort
8 #include <functional> // std::less
9 #include <iostream> // std::cout and std::cerr
10 #include <iterator> // std::iterator_traits
11 #include "time.h"
12
13 template <typename position>
14 void show(position a, position e) {
15   while (a != e) {
16     std::cout << int(*a) << " ";
17     ++a;
18   }
19   std::cout << std::endl;
20   std::cout.flush();
21 }
22
23 typedef int E;
24 typedef std::allocator<E> A;
25
26 #include "data_structure.i++" // defines X using E and A
27
28 template <typename position>
29 void generate(position p, position r, char z) {
30   typedef typename std::iterator_traits<position>::value_type element;
31   switch (z) {
32   case 'd':
33     for (position q = p; q != r; ++q)
34       *q = element((r - 1) - q);
35     break;
36   case 'i':
37     for (position q = p; q != r; ++q)
38       *q = element(q - p);
39     break;
40   case 'r':
41     for (position q = p; q != r; ++q)
42       *q = element(q - p);
43     std::random_shuffle(p, r);
44     break;
45   case 'b':
46     bool t = false;
47     for (position q = p; q != r; ++q) {
48       *q = element(t);
49       t = !t;
50     }
51     break;
52   }
53 }
```

```
54
55 void usage(int argc, char **argv) {
56   std::cerr << "Usage: " << argv[0]
57             << " <n> <'i'ncreasing | 'd'ecreasing | 'r'andom | 'b'ool>"
58             << std::endl;
59   exit(1);
60 }
61
62 int main(int argc, char** argv) {
63   typedef std::less<E> C;
64
65   unsigned int n;
66   char method;
67   if (argc == 1) {
68     n = 15;
69     method = 'i';
70   }
71   else if (argc == 2) {
72     n = atoi(argv[1]);
73     method = 'i';
74   }
75   else if (argc != 3) {
76     usage(argc, argv);
77   }
78   else {
79     n = atoi(argv[1]);
80     method = *argv[2];
81   }
82   if (n < 1 || n > MAXSIZE) {
83     std::cerr << "n out of bounds [1.."
84               << MAXSIZE
85               << "]"
86               << std::endl;
87     usage(argc, argv);
88   };
89   switch (method) {
90   case 'd':
91   case 'i':
92   case 'r':
93   case 'b':
94     break;
95   default:
96     std::cerr << "Method not in ['d','i','r','b']" << std::endl;
97     usage(argc, argv);
98   }
99
100  E* const  a = new E[n];
101  generate(a, a + n, method);
102
103  std::vector<X> v;
104  v.resize(MAXSIZE / n);
105  for (unsigned int t = 0; t < MAXSIZE / n; ++t) {
106    for (unsigned int i = 0; i != n; ++i) {
107      v[t].push_back(a[i]);
108    }
109  }
110
111  clock_t clock_start = clock();
112  for (volatile unsigned int t = 0; t != MAXSIZE / n; ++t)  {
113    auto c = v[t].begin();
114    auto e = v[t].end();
115    for (; c != e; ++c) {
116      *c = E(0);
117    }
118  }
```

```
119   clock_t clock_stop = clock();
120
121   for (unsigned int t = 0; t < MAXSIZE / n; ++t) {
122     for (unsigned int i = n; i != 0; ) {
123       --i;
124       v[t].pop_back();
125     }
126   }
127
128   long long t = MAXSIZE / n;
129   t *= n;
130   double seconds = (clock_stop - clock_start) / (double) CLOCKS_PER_SEC;
131   std::cout.precision(3);
132   std::cout << n << '\t' << 1.0e9 * seconds / double(t) << std::endl;
133
134   delete[] a;
135   return 0;
136 }
```

§ 19 *jump-driver.c++*

```
1  #if ! defined(MAXSIZE)
2
3  #define MAXSIZE (64*1024*1024)
4
5  #endif
6
7  #include <algorithm>  // std::random_shuffle, std::make_heap, std::sort
8  #include <functional> // std::less
9  #include <iostream> // std::cout and std::cerr
10 #include <iterator> // std::iterator_traits
11 #include "time.h"
12
13 template <typename position>
14 void show(position a, position e) {
15   while (a != e) {
16     std::cout << int(*a) << " ";
17     ++a;
18   }
19   std::cout << std::endl;
20   std::cout.flush();
21 }
22
23 typedef int E;
24 typedef std::allocator<E> A;
25
26 #include "data-structure.i++" // defines X using E and A
27
28 template <typename position>
29 void generate(position p, position r, char z) {
30   typedef typename std::iterator_traits<position>::value_type element;
31   switch (z) {
32   case 'd':
33     for (position q = p; q != r; ++q)
34       *q = element((r - 1) - q);
35     break;
36   case 'i':
37     for (position q = p; q != r; ++q)
38       *q = element(q - p);
39     break;
40   case 'r':
41     for (position q = p; q != r; ++q)
42       *q = element(q - p);
43     std::random_shuffle(p, r);
```

```
44      break;
45    case 'b':
46      bool t = false;
47      for (position q = p; q != r; ++q) {
48        *q = element(t);
49        t = !t;
50      }
51      break;
52    }
53  }
54
55  void usage(int argc, char **argv) {
56    std::cerr << "Usage: " << argv[0]
57              << " <n> <'i'ncreasing | 'd'ecreasing | 'r'andom | 'b'ool>"
58              << std::endl;
59    exit(1);
60  }
61
62  int main(int argc, char** argv) {
63    typedef std::less<E> C;
64
65    unsigned int n;
66    char method;
67    if (argc == 1) {
68      n = 15;
69      method = 'i';
70    }
71    else if (argc == 2) {
72      n = atoi(argv[1]);
73      method = 'i';
74    }
75    else if (argc != 3) {
76      usage(argc, argv);
77    }
78    else {
79      n = atoi(argv[1]);
80      method = *argv[2];
81    }
82    if (n < 1 || n > MAXSIZE) {
83      std::cerr << "n out of bounds [1.."
84                << MAXSIZE
85                << "]"
86                << std::endl;
87      usage(argc, argv);
88    };
89    switch (method) {
90    case 'd':
91    case 'i':
92    case 'r':
93    case 'b':
94      break;
95    default:
96      std::cerr << "Method not in ['d','i','r','b']" << std::endl;
97      usage(argc, argv);
98    }
99
100   E* const  a = new E[n];
101   generate(a, a + n, method);
102
103   std::vector<X> v;
104   v.resize(MAXSIZE / n);
105   for (unsigned int t = 0; t < MAXSIZE / n; ++t) {
106     for (unsigned int i = 0; i != n; ++i) {
107       v[t].push_back(a[i]);
108     }
```

```
109   }
110
111   unsigned int const mask = n - 1;
112   clock_t clock_start = clock();
113   for (volatile unsigned int t = 0; t != MAXSIZE / n; ++t)  {
114     auto c = v[t].begin();
115     if (c != v[t].end()) {
116       *c = E(0);
117     }
118     for (unsigned int i = prime; i != 0; i = (i + prime) & mask) {
119       *(c + i) = E(0);
120     }
121   }
122   clock_t clock_stop = clock();
123
124   for (unsigned int t = 0; t < MAXSIZE / n; ++t) {
125     for (unsigned int i = n; i != 0; ) {
126       --i;
127       v[t].pop_back();
128     }
129   }
130
131   long long t = MAXSIZE / n;
132   t *= n;
133   double seconds = (clock_stop - clock_start) / (double) CLOCKS_PER_SEC;
134   std::cout.precision(3);
135   std::cout << n << '\t' << 1.0e9 * seconds / double(t) << std::endl;
136
137   delete[] a;
138   return 0;
139 }
```

§ 20 *sort-driver.c++*

```
1  #if ! defined(MAXSIZE)
2
3  #define MAXSIZE (64*1024*1024)
4
5  #endif
6
7  #include <algorithm>  // std::random_shuffle, std::make_heap, std::sort
8  #include <cmath> // ilogb
9  #include <functional> // std::less
10 #include <iostream> // std::cout and std::cerr
11 #include <iterator> // std::iterator_traits
12 #include "time.h"
13
14 extern int ilogb(double) throw();
15
16 template <typename position>
17 void show(position a, position e) {
18   while (a != e) {
19     std::cout << int(*a) << " ";
20     ++a;
21   }
22   std::cout << std::endl;
23   std::cout.flush();
24 }
25
26 template <typename position>
27 bool is_permutation(position first, position beyond) {
28   typedef typename std::iterator_traits<position>::value_type element;
29   std::sort(first, beyond);
```

```
30   for (position q = first; q != beyond; ++q) {
31     element i = element(q − first);
32     if (*q != i) {
33       std::cerr << i << ": element missing " << *q << " instead" << std::endl;
34       std::cerr << "n: " << beyond − first << std::endl;
35       return false;
36     }
37   }
38   return true;
39 }
40
41 template <class position, class ordering>
42 bool is_sorted(position first, position beyond, ordering less) {
43   typedef typename std::iterator_traits<position>::difference_type index;
44   const position a = first − 1;
45   const index n = beyond − first;
46   bool violated = false;
47   for (index i = n; i > 1; i−−)
48     if (less(a[i], a[i − 1])) {
49       std::cerr << i << ": me " << a[i] << "; before " << a[i − 1] << std::endl;
50       violated = true;
51     }
52   return ! violated;
53 }
54
55 typedef int E;
56 typedef std::allocator<E> A;
57
58 #include "data–structure.i++" // defines X using E and A
59
60 template <typename position>
61 void generate(position p, position r, char z) {
62   typedef typename std::iterator_traits<position>::value_type element;
63   switch (z) {
64   case 'd':
65     for (position q = p; q != r; ++q)
66       *q = element((r − 1) − q);
67     break;
68   case 'i':
69     for (position q = p; q != r; ++q)
70       *q = element(q − p);
71     break;
72   case 'r':
73     for (position q = p; q != r; ++q)
74       *q = element(q − p);
75     std::random_shuffle(p, r);
76     break;
77   case 'b':
78     bool t = false;
79     for (position q = p; q != r; ++q) {
80       *q = element(t);
81       t = !t;
82     }
83     break;
84   }
85 }
86
87 void usage(int argc, char **argv) {
88   std::cerr << "Usage: " << argv[0]
89             << " <N> <'i'ncreasing | 'd'ecreasing | 'r'andom | 'b'ool>"
90             << std::endl;
91   exit(1);
92 }
93
94 int main(int argc, char** argv) {
```

```
95  typedef std::less<E> C;
96
97  unsigned int n;
98  char method;
99  if (argc == 1) {
100   n = 15;
101   method = 'i';
102 }
103 else if (argc == 2) {
104   n = atoi(argv[1]);
105   method = 'r';
106 }
107 else if (argc != 3) {
108   usage(argc, argv);
109 }
110 else {
111   n = atoi(argv[1]);
112   method = *argv[2];
113 }
114 if (n < 1 || n > MAXSIZE) {
115   std::cerr << "n out of bounds [1.."
116             << MAXSIZE
117             << "]"
118             << std::endl;
119   usage(argc, argv);
120 };
121 switch (method) {
122 case 'd':
123 case 'i':
124 case 'r':
125 case 'b':
126   break;
127 default:
128   std::cerr << "Method not in ['d','i','r','b']" << std::endl;
129   usage(argc, argv);
130 }
131
132 E* const  a = new E[MAXSIZE];
133 auto d = a;
134 for (unsigned int t = 0; t != MAXSIZE / n; ++t) {
135   generate(d, d + n, method);
136   d = d + n;
137 }
138
139 X b(MAXSIZE); //   X b(MAXSIZE); for segment.h++ and hashed_array_tree.h++
140 for (unsigned int i = 0; i != MAXSIZE; ++i) {
141   b.push_back(a[i]);
142 }
143
144 auto c = b.begin();
145 clock_t clock_start = clock();
146 for (volatile unsigned int t = 0; t != MAXSIZE / n; ++t)  {
147   std::sort(c, c + n, C());
148   c = c + n;
149 }
150 clock_t clock_stop = clock();
151
152 for (volatile unsigned int i = MAXSIZE; i > 0;) {
153   --i;
154   a[i] = b[i];
155   b.pop_back();
156 }
157
158 d = a;
159 for (volatile unsigned int t = 0; t != MAXSIZE / n; ++t) {
```

```
160     bool ok = ::is_sorted(d, d + n, C());
161     if (! ok) {
162       return 1;
163     }
164     if (method == 'd' || method == 'i' || method == 'r') {
165       ok = ::is_permutation(d, d + n);
166       if (! ok) {
167         show(d, d + n);
168         return 2;
169       }
170     }
171     d = d + n;
172   }
173
174   long long t = MAXSIZE / n;
175   t *= n * ilogb(n);
176   double seconds = (clock_stop - clock_start) / (double) CLOCKS_PER_SEC;
177   std::cout.precision(3);
178   std::cout << n << '\t' << 1.0e9 * seconds / double(t) << std::endl;
179
180   delete[] a;
181   return 0;
182 }
```

§ 21 *grow-driver.c++*

```
1  #if ! defined(MAXSIZE)
2
3  #define MAXSIZE (64*1024*1024)
4
5  #endif
6
7  #include <algorithm>  // std::random_shuffle, std::make_heap, std::sort
8  #include <functional> // std::less
9  #include <iostream> // std::cout and std::cerr
10 #include <iterator> // std::iterator_traits
11 #include "time.h"
12
13 template <typename position>
14 void show(position a, position e) {
15   while (a != e) {
16     std::cout << int(*a) << " ";
17     ++a;
18   }
19   std::cout << std::endl;
20   std::cout.flush();
21 }
22
23 typedef int E;
24 typedef std::allocator<E> A;
25
26 #include "data_structure.i++" // defines X using E and A
27
28 template <typename position>
29 void generate(position p, position r, char z) {
30   typedef typename std::iterator_traits<position>::value_type element;
31   switch (z) {
32   case 'd':
33     for (position q = p; q != r; ++q)
34       *q = element((r - 1) - q);
35     break;
36   case 'i':
37     for (position q = p; q != r; ++q)
38       *q = element(q - p);
```

```
39      break;
40    case 'r':
41      for (position q = p; q != r; ++q)
42        *q = element(q - p);
43      std::random_shuffle(p, r);
44      break;
45    case 'b':
46      bool t = false;
47      for (position q = p; q != r; ++q) {
48        *q = element(t);
49        t = !t;
50      }
51      break;
52    }
53  }
54
55  void usage(int argc, char **argv) {
56    std::cerr << "Usage: " << argv[0]
57              << " <n> <'i'ncreasing | 'd'ecreasing | 'r'andom | 'b'ool>"
58              << std::endl;
59    exit(1);
60  }
61
62  int main(int argc, char** argv) {
63    typedef std::less<E> C;
64
65    unsigned int n;
66    char method;
67    if (argc == 1) {
68      n = 15;
69      method = 'i';
70    }
71    else if (argc == 2) {
72      n = atoi(argv[1]);
73      method = 'i';
74    }
75    else if (argc != 3) {
76      usage(argc, argv);
77    }
78    else {
79      n = atoi(argv[1]);
80      method = *argv[2];
81    }
82    if (n < 1 || n > MAXSIZE) {
83      std::cerr << "n out of bounds [1.."
84                << MAXSIZE
85                << "]"
86                << std::endl;
87      usage(argc, argv);
88    };
89    switch (method) {
90    case 'd':
91    case 'i':
92    case 'r':
93    case 'b':
94      break;
95    default:
96      std::cerr << "Method not in ['d','i','r','b']" << std::endl;
97      usage(argc, argv);
98    }
99
100   E* const  a = new E[n];
101   generate(a, a + n, method);
102
103   std::vector<X> v;
```

```
104   v.resize(MAXSIZE / n);
105
106   clock_t clock_start = clock();
107   for (unsigned int t = 0; t < MAXSIZE / n; ++t) {
108     for (unsigned int i = 0; i != n; ++i) {
109       v[t].push_back(a[i]);
110     }
111   }
112   clock_t clock_stop = clock();
113
114   for (unsigned int t = 0; t < MAXSIZE / n; ++t) {
115     for (unsigned int i = n; i != 0; ) {
116       --i;
117       v[t].pop_back();
118     }
119   }
120
121   long long t = MAXSIZE / n;
122   t *= n;
123   double seconds = (clock_stop - clock_start) / (double) CLOCKS_PER_SEC;
124   std::cout.precision(3);
125   std::cout << n << '\t' << 1.0e9 * seconds / double(t) << std::endl;
126
127   delete[] a;
128   return 0;
129 }
```

§ 22 *shrink-driver.c++*

```
1  #if ! defined(MAXSIZE)
2
3  #define MAXSIZE (64*1024*1024)
4
5  #endif
6
7  #include <algorithm>  // std::random_shuffle, std::make_heap, std::sort
8  #include <functional> // std::less
9  #include <iostream> // std::cout and std::cerr
10 #include <iterator> // std::iterator_traits
11 #include "time.h"
12
13 template <typename position>
14 void show(position a, position e) {
15   while (a != e) {
16     std::cout << int(*a) << " ";
17     ++a;
18   }
19   std::cout << std::endl;
20   std::cout.flush();
21 }
22
23 typedef int E;
24 typedef std::allocator<E> A;
25
26 #include "data_structure.i++" // defines X using E and A
27
28 template <typename position>
29 void generate(position p, position r, char z) {
30   typedef typename std::iterator_traits<position>::value_type element;
31   switch (z) {
32   case 'd':
33     for (position q = p; q != r; ++q)
34       *q = element((r - 1) - q);
35     break;
```

```
36    case 'i':
37      for (position q = p; q != r; ++q)
38        *q = element(q - p);
39      break;
40    case 'r':
41      for (position q = p; q != r; ++q)
42        *q = element(q - p);
43      std::random_shuffle(p, r);
44      break;
45    case 'b':
46      bool t = false;
47      for (position q = p; q != r; ++q) {
48        *q = element(t);
49        t = !t;
50      }
51      break;
52    }
53  }
54
55  void usage(int argc, char **argv) {
56    std::cerr << "Usage: " << argv[0]
57              << " <n> <'i'ncreasing | 'd'ecreasing | 'r'andom | 'b'ool>"
58              << std::endl;
59    exit(1);
60  }
61
62  int main(int argc, char** argv) {
63    typedef std::less<E> C;
64
65    unsigned int n;
66    char method;
67    if (argc == 1) {
68      n = 15;
69      method = 'i';
70    }
71    else if (argc == 2) {
72      n = atoi(argv[1]);
73      method = 'i';
74    }
75    else if (argc != 3) {
76      usage(argc, argv);
77    }
78    else {
79      n = atoi(argv[1]);
80      method = *argv[2];
81    }
82    if (n < 1 || n > MAXSIZE) {
83      std::cerr << "n out of bounds [1.."
84                << MAXSIZE
85                << "]"
86                << std::endl;
87      usage(argc, argv);
88    };
89    switch (method) {
90    case 'd':
91    case 'i':
92    case 'r':
93    case 'b':
94      break;
95    default:
96      std::cerr << "Method not in ['d','i','r','b']" << std::endl;
97      usage(argc, argv);
98    }
99
100   E* const  a = new E[n];
```

```
101   generate(a, a + n, method);
102
103   std::vector<X> v;
104   v.resize(MAXSIZE / n);
105   for (unsigned int t = 0; t < MAXSIZE / n; ++t) {
106     for (unsigned int i = 0; i != n; ++i) {
107       v[t].push_back(a[i]);
108     }
109   }
110
111   clock_t clock_start = clock();
112   for (unsigned int t = 0; t < MAXSIZE / n; ++t) {
113     for (unsigned int i = n; i != 0; ) {
114       --i;
115       v[t].pop_back();
116     }
117   }
118   clock_t clock_stop = clock();
119
120   long long t = MAXSIZE / n;
121   t *= n;
122   double seconds = (clock_stop - clock_start) / (double) CLOCKS_PER_SEC;
123   std::cout.precision(3);
124   std::cout << n << '\t' << 1.0e9 * seconds / double(t) << std::endl;
125
126   delete[] a;
127   return 0;
128 }
```

§ 23 *space-driver.c++*

```
1  /*
2     Measures the amount of space taken up by a data structure
3
4     Authors: Jyrki Katajainen, Bjarke Buur Mortensen © 2001, 2012
5  */
6
7  #include <cstddef> // std::size_t
8  #include <cstdlib>
9  #include <iomanip>
10 #include <iostream>
11
12 #include "counting_allocator.h++"
13
14 typedef int E;
15 typedef counting_allocator<E> A;
16
17 #include "data_structure.i++" // defines X using E and A
18
19 float run(std::size_t n) {
20   counting_allocator_base::reset_counts();
21   X container;
22   for (std::size_t i = 0; i != n; ++i) {
23     container.push_back(random());
24   }
25   return float(counting_allocator_base::bytes_in_use());
26 }
27
28 int main(int argc, char* argv[]) {
29   srand(1837362);
30   for (std::size_t n = 100000; n <= 10000000; n += 100000) {
31     std::cout.setf(std::ios::fixed, std::ios::floatfield);
32     std::cout.precision(3);
33     float bytes_in_use = run(n);
```

```
34      float overhead = bytes_in_use - float(n * sizeof(E));
35      float in_procents = 100 * overhead / (n * sizeof(E));
36      std::cout << n << "\t" << in_procents << std::endl;
37    }
38    return 0;
39  }
```

Makefile

§ 24 *makefile*

```
1  CXX=g++-4.7
2  CXXFLAGS=-O3 -Wall -std=c++11 -msse4.2 -mabm
3  TESTFLAGS=-g -Wall -x c++ -std=c++11 -msse4.2 -mabm
4  INCLUDES=-I ./CCF
5
6  vector_files:= $(wildcard *.i++)
7  vector_bases:= $(basename $(vector_files))
8  vector_test:= $(addsuffix .vector_test, $(vector_bases))
9  sort:= $(addsuffix .sort, $(vector_bases))
10 scan:= $(addsuffix .scan, $(vector_bases))
11 jump:= $(addsuffix .jump, $(vector_bases))
12 grow:= $(addsuffix .grow, $(vector_bases))
13 shrink:= $(addsuffix .shrink, $(vector_bases))
14 space:= $(addsuffix .space, $(vector_bases))
15
16 queue_files:= binomial_queue.i++
17 queue_bases:= $(basename $(queue_files))
18 queue_test:= $(addsuffix .queue_test, $(queue_bases))
19 push:= $(addsuffix .push, $(queue_bases))
20 borrow:= $(addsuffix .borrow, $(queue_bases))
21 erase:= $(addsuffix .erase, $(queue_bases))
22 pop:= $(addsuffix .pop, $(queue_bases))
23
24 N = 1024 32768 1048576 33554432
25
26 $(vector_test): %.vector_test : %.i++
27         @cp $*.i++ data_structure.i++
28         $(CXX) $(TESTFLAGS) $(INCLUDES) vector_test_driver.c++
29         @for n in 10 20 30 40 50 ; do \
30           echo $$n ; \
31           ./a.out $$n ; \
32         done;
33
34 $(queue_test): %.queue_test : %.i++
35         @cp $*.i++ data_structure.i++
36         $(CXX) $(TESTFLAGS) queue_test_driver.c++
37         @for n in 10 20 30 40 50 ; do \
38           echo $$n ; \
39           ./a.out $$n ; \
40         done;
41
42 $(sort): %.sort : %.i++
43         @cp $*.i++ data_structure.i++
44         $(CXX) $(CXXFLAGS) sort_driver.c++
45         @for n in $(N) ; do \
46           ./a.out $$n ; \
47         done; \
48         rm -f ./a.out
49
50 $(scan): %.scan : %.i++
51         @cp $*.i++ data_structure.i++
52         $(CXX) $(CXXFLAGS) scan_driver.c++
53         @for n in $(N) ; do \
```

```
54            ./a.out $$n ; \
55          done; \
56          rm -f ./a.out
57
58  $(jump): %.jump : %.i++
59          @cp $*.i++ data_structure.i++
60          $(CXX) $(CXXFLAGS) jump_driver.c++
61          @for n in $(N) ; do \
62            ./a.out $$n ; \
63          done; \
64          rm -f ./a.out
65
66  $(grow): %.grow : %.i++
67          @cp $*.i++ data_structure.i++
68          $(CXX) $(CXXFLAGS) grow_driver.c++
69          @for n in $(N) ; do \
70            ./a.out $$n ; \
71          done; \
72          rm -f ./a.out
73
74  $(shrink): %.shrink : %.i++
75          @cp $*.i++ data_structure.i++
76          $(CXX) $(CXXFLAGS) shrink_driver.c++
77          @for n in $(N) ; do \
78            ./a.out $$n ; \
79          done; \
80          rm -f ./a.out
81
82  $(space): %.space : %.i++
83          @echo "#" $* "space_driver.c++"
84          @cp $*.i++ data_structure.i++
85          @$(CXX) $(CXXFLAGS) space_driver.c++
86          @./a.out;
87          @rm -f ./a.out
88
89  $(push): %.push : %.i++
90          @cp $*.i++ data_structure.i++
91          $(CXX) $(CXXFLAGS) push_driver.c++
92          @for n in $(N) ; do \
93            ./a.out $$n ; \
94          done; \
95          rm -f ./a.out
96
97  $(borrow): %.borrow : %.i++
98          @cp $*.i++ data_structure.i++
99          $(CXX) $(CXXFLAGS) borrow_driver.c++
100         @for n in $(N) ; do \
101           ./a.out $$n ; \
102         done; \
103         rm -f ./a.out
104
105 $(erase): %.erase : %.i++
106         @cp $*.i++ data_structure.i++
107         $(CXX) $(CXXFLAGS) erase_driver.c++
108         @for n in $(N) ; do \
109           ./a.out $$n ; \
110         done; \
111         rm -f ./a.out
112
113 $(pop): %.pop : %.i++
114         @cp $*.i++ data_structure.i++
115         $(CXX) $(CXXFLAGS) pop_driver.c++
116         @for n in $(N) ; do \
117           ./a.out $$n ; \
118         done; \
```

```
119         rm -f ./a.out
120
121 factory-test:
122         $(CXX) $(TESTFLAGS) -DUNITTEST_FACTORY factory.h++
123         ./a.out
124
125 module-test:
126         $(CXX) $(TESTFLAGS) -DUNITTEST_COLLECTION_USE_CASE collection-use-case.m
                ++
127         ./a.out
128
129 clean:
130         - rm  -f temp core a.out data-structure.i++ *.o 2>/dev/null
131
132 veryclean: clean
133         - rm -f *~ */*~ 2>/dev/null
134
135 find:
136         find . -type f -print -exec grep $(word) {} \; | less # or -name '*.cc'
```