Dynamic-array kernels

Katajainen, Jyrki

Publication date:
2016

Document version
Publisher's PDF, also known as Version of record

Citation for published version (APA):
Katajainen, J. (2016). *Dynamic-array kernels*. Department of Computer Science, University of Copenhagen. CPH STL Report, No. 2, Vol.. 2016

Dynamic-Array Kernels

Jyrki Katajainen

Department of Computer Science, University of Copenhagen
Universitetsparken 5, 2100 Copenhagen East, Denmark
jyrki@di.ku.dk

Abstract. This report together with an accompanying `tar` ball contains the programs used in the benchmarks discussed in the paper "Worst-Case-Efficient Dynamic Arrays in Practice". With this source code it should be possible for others to reproduce the reported results.

Keywords. Data structures, dynamic arrays, worst-case efficiency

CPH STL Report 2016-2, March 2016.

Release date

2016-03-28

Kernels

The basic operations of a dynamic array are **operator**[], `push_back`, and `pop_back`. The study "Worst-Case-Efficient Dynamic Arrays in Practice" was an examination of variations of dynamic arrays that support these operations at $O(1)$ worst-case cost. In that paper the dynamic-array kernels considered were characterized as follows:

`std::vector`**:** This was the standard-library implementation that shipped with our `g++` compiler (version 4.8.4). It stored the values in one segment, `push_back` relied on doubling, and `pop_back` was a noop—memory was released only at the time of destruction. Compared to the other alternatives, this version only supported `push_back` at $O(1)$ amortized cost.

`cphstl::resizable_array`**:** This solution relied on doubling, halving, and incremental copying.

`cphstl::pile`**:** This version implemented the level-wise-allocated pile described in [4]. The data was split into a logarithmic number of contiguous segments, values were not moved due to reorganizations, and the three operations of interest were all supported at $O(1)$ worst-case cost.

`cphstl::sliced_array`**:** This version imitated the standard-library implementation of a double-ended queue. It was like a page table where the directory was implemented as a resizable array and the pages (memory segments) were arrays of fixed capacity (512 values).

`cphstl::space_efficient_array`**:** This version was as the block-wise-allocated pile described in [4], but the implementation was simplified by seeing it as a pile of hashed array trees [6]. This version matched the space and time bounds proved to be optimal in [**?**].

Architecture

In the CPH STL [3], the container classes have been implemented using the bridge design pattern [7, Sect. 14.4]. Above the dynamic-array kernels, two application program interfaces (APIs) are available: `cphstl`::`vector` which provides the same functionality as `std`::`vector` from the C++ standard library [2, Clause 23.3.6], and `cphleda`::`array` which provides the same functionality as `leda`::`array` from the LEDA library [1]. Our LEDA APIs are discussed in length in [5].

An architectural overview of the dynamic-array part of the library is given in Fig. 1 in the form of a UML class diagram. The full allocator interface is described in the C++ standard [2, Clause 20.7.9]. Any of the kernels can be given as a template argument for both APIs. On purpose, the kernels have not been made minimal so they can also be used individually. The rank-iterator class template can be used to get both mutable and immutable iterators for all kernels. Based on the actual needs, a user can tailor the

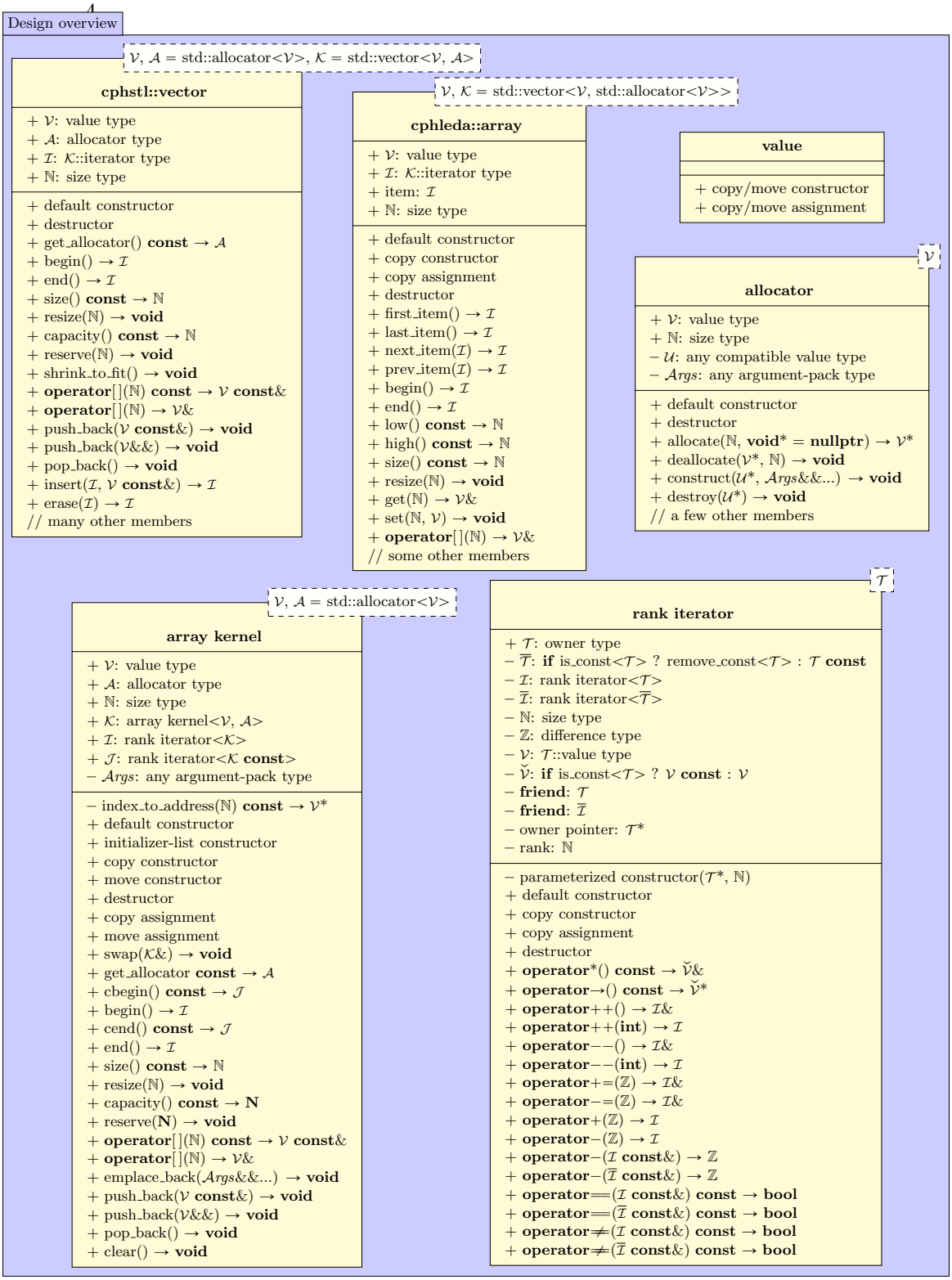

Figure 1. A UML class diagram illustrating the design of the dynamic-array part of the CPH STL; + means that a member is public and − that it is private

components to meet the desired safety and complexity requirements. A serious user can even build his own kernel and use it with the existing APIs. This kind of customization and easy of extension have been the main design objectives while creating the CPH STL.

Contents

References

[1] Algorithmic Solutions Software GmbH, The LEDA User Manual, version 6.4, World-wide Web Document (2008). Available at `http://www.algorithmic-solutions.com/leda/index.htm`.

[2] The C++ Standards Committee, Standard for Programming Language C++, Working Draft **N4296**, ISO/IEC, Geneva (2014).

[3] Department of Computer Science, University of Copenhagen, The CPH STL, Website accessible at `http://cphstl.dk/` (2000–2016).

[4] J. Katajainen and B. B. Mortensen, Experiences with the design and implementation of space-efficient deques, *Proceedings of the 5th Workshop on Algorithm Engineering*, *Lecture Notes in Computer Science* **2141**, Springer, Berlin/Heidelberg (2001), 39–50.

[5] M. Neidhardt and B. Simonsen, Extending the CPH STL with LEDA APIs, CPH STL Report **2009-8**, Department of Computer Science, University of Copenhagen, Copenhagen (2009).

[6] E. Sitarski, Algorithm alley: HATs: Hashed array trees: Fast variable-length arrays, *Dr. Dobb's Journal* **21**, 11 (1996).

[7] D. Vandervoorde and N. M. Josuttis, *C++ Templates: The Complete Guide*, Pearson Education, Inc., Boston (2003).

Kernels

contiguous_array.h++

```
/*
  This is an experimental array which stores the elements contiguosly
  and has a fixed capacity. The capacity must be set using the
  reserve() function.

  Author: Jyrki Katajainen @ 2012, 2016
*/

#include <cassert> // assert macro
#include <cstddef> // std::size_t and std::ptrdiff_t
#include <memory> // std::allocator, std::uninitialized_copy/fill
#include <utility> // std::forward, std::move, std::swap

namespace cphstl {

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

    using value_type = V;
    using allocator_type = typename A::template rebind<V>::other;
    using reference = V&;
    using const_reference = V const&;
    using pointer = V*;
    using const_pointer = V const*;
    using difference_type = std::ptrdiff_t;
    using size_type = std::size_t;
    using iterator = V*;
    using const_iterator = V const*;

  protected:

    allocator_type allocator;
    V* X;
    size_type X_size;
    size_type X_capacity;

  public:

    explicit contiguous_array(A const& a = A())
      : allocator(a), X(nullptr), X_size(0), X_capacity(0) {
    }

    contiguous_array(contiguous_array const& other)
      : contiguous_array(other.allocator) {
      V* tmp = nullptr;
      size_type n = other.capacity();
      if (n != 0) {
```

```
49        tmp = allocator.allocate(n);
50        std::uninitialized_copy(other.X, other.X + n, tmp);
51      }
52      X = tmp;
53      X_size = other.size();
54      X_capacity = n;
55    }
56
57    contiguous_array(contiguous_array&& other)
58      : contiguous_array(other.allocator) {
59      swap(other);
60    }
61
62    ~contiguous_array() {
63      clear();
64    }
65
66    contiguous_array& operator=(contiguous_array const& other) {
67      if (this ≠ &other) {
68        contiguous_array tmp(other);
69        swap(tmp);
70      }
71      return *this;
72    }
73
74    contiguous_array& operator=(contiguous_array&& other) {
75      if (this ≠ &other) {
76        contiguous_array tmp(std::move(other));
77        swap(tmp);
78      }
79      return *this;
80    }
81
82    void swap(contiguous_array& other) {
83      std::swap(allocator, other.allocator);
84      std::swap(X, other.X);
85      std::swap(X_size, other.X_size);
86      std::swap(X_capacity, other.X_capacity);
87    }
88
89    const_iterator begin() const {
90      return X;
91    }
92
93    iterator begin() {
94      return X;
95    }
96
97    const_iterator end() const {
98      return X + X_size;
99    }
100
```

```
101     iterator end() {
102       return X + X_size;
103     }
104
105     V* data() noexcept {
106       return X;
107     }
108
109     V const* data() const noexcept {
110       return X;
111     }
112
113     allocator_type get_allocator() const {
114       return allocator;
115     }
116
117     size_type size() const {
118       return X_size;
119     }
120
121     void resize(size_type n) {
122       assert(n ≤ capacity());
123       if (n ≤ X_size) {
124         for (size_type i = n; i < X_size; ++i) {
125           allocator.destroy(X + i);
126         }
127       }
128       else {
129         std::uninitialized_fill(X + X_size, X + n, V());
130       }
131       X_size = n;
132     }
133
134     size_type capacity() const {
135       return X_capacity;
136     }
137
138     void reserve(size_type n) {
139       assert(X == nullptr); // capacity can only be set once
140       X = allocator.allocate(n);
141       X_size = 0;
142       X_capacity = n;
143     }
144
145     const_reference operator[](size_type i) const {
146       return X[i];
147     }
148
149     reference operator[](size_type i) {
150       return X[i];
151     }
152
```

```
153     template <typename... Args>
154     void emplace_back(Args&&... args) {
155       assert(X_size < X_capacity);
156       new (X + X_size) V(std::forward<Args>(args)...);
157       ++X_size;
158     }
159
160     void push_back(V const& x) {
161       assert(X_size < X_capacity);
162       allocator.construct(X + X_size, x);
163       ++X_size;
164     }
165
166     void push_back(V&& x) {
167       assert(X_size < X_capacity);
168       allocator.construct(X + X_size, std::move(x));
169       ++X_size;
170     }
171
172     void pop_back() {
173       assert(X_size != 0);
174       allocator.destroy(X + X_size);
175       --X_size;
176     }
177
178     void clear() {
179       for (size_type i = 0; i != X_size; ++i) {
180         allocator.destroy(X + i);
181       }
182       X_size = 0;
183       if (X != nullptr) {
184         allocator.deallocate(X, X_capacity);
185       }
186       X = nullptr;
187       X_capacity = 0;
188     }
189   };
190 }
```

resizable_array.h++

```
1  /*
2    This is an experimental implementation of a resizable array based
3    on doubling, halving, and incremental copying.
4
5    Author: Jyrki Katajainen © 2012, 2013, 2015, 2016
6  */
7
8  #ifndef __CPHSTL_RESIZABLE_ARRAY__
9  #define __CPHSTL_RESIZABLE_ARRAY__
10
11 #include <algorithm> // std::max
```

```
#include <cassert> // assert macro
#include <cstddef> // std::size_t, std::ptrdiff_t
#include <initializer_list> // std::initializer_list
#include <iterator> // std::make_move_iterator
#include <memory> // std::allocator, std::uninitialized_copy
#include "rank-iterator.h++" // cphstl::rank_iterator
#include <utility> // std::forward, std::move, std::swap

namespace cphstl {

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

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

  private:

    V* data() noexcept;
    V const* data() const noexcept;

  protected:

    allocator_type allocator;
    V* X;
    V* Y;
    size_type X_size;
    size_type Y_size;
    size_type X_capacity;
    size_type Y_capacity;

    V* index_to_address(size_type rank) const {
      if (rank < X_size) {
        return X + rank;
      }
      return Y + rank;
    }

    template <typename I>
    void append(I first, I past) {
      I p = first;
      I q = first;
```

```
64      try {
65        for ( ; q ≠ past; ++q) {
66          push_back(*q);
67        }
68      }
69      catch (...) {
70        for ( ; p ≠ q; ++p) {
71          pop_back();
72        }
73        throw;
74      }
75    }
76
77    void append_n(size_type k, V const& value) {
78      size_type i = 0;
79      size_type j = 0;
80      try {
81        for ( ; j ≠ k; ++j) {
82          push_back(value);
83        }
84      }
85      catch (...) {
86        for ( ; i ≠ j; ++i) {
87          pop_back();
88        }
89      }
90    }
91
92  public:
93
94    explicit resizable_array(A const& a = A())
95      : allocator(a), X(nullptr), Y(nullptr), X_size(0), Y_size(0),
96        X_capacity(0), Y_capacity(0) {
97      X = allocator.allocate(1);
98      X_capacity = 1;
99    }
100
101   resizable_array(std::initializer_list<V> s)
102     : resizable_array(A()) {
103     append(s.begin(), s.end());
104   }
105
106   resizable_array(size_type n, A const& a)
107     : resizable_array(a) {
108     append_n(n, V());
109   }
110
111   resizable_array(resizable_array const& other)
112     : resizable_array(other.get_allocator()) {
113     V* tmp = nullptr;
114     size_type n = other.capacity();
115     tmp = allocator.allocate(n);
```

```
116       std::uninitialized_copy(other.begin(), other.begin() + n, tmp);
117       X = tmp;
118       X_size = other.size();
119       X_capacity = n;
120     }
121
122     resizable_array(resizable_array&& other)
123       : resizable_array(other.get_allocator()) {
124       swap(other);
125     }
126
127     ~resizable_array() {
128       clear();
129       if (X != nullptr) {
130         allocator.deallocate(X, X_capacity);
131       }
132     }
133
134     resizable_array& operator=(resizable_array const& other) {
135       if (this != &other) {
136         resizable_array tmp(other);
137         swap(tmp);
138       }
139       return *this;
140     }
141
142     resizable_array& operator=(resizable_array&& other) {
143       if (this != &other) {
144         resizable_array tmp(std::move(other));
145         swap(tmp);
146       }
147       return *this;
148     }
149
150     void swap(resizable_array& other) {
151       std::swap(allocator, other.allocator);
152       std::swap(X, other.X);
153       std::swap(X_size, other.X_size);
154       std::swap(X_capacity, other.X_capacity);
155       std::swap(Y, other.Y);
156       std::swap(Y_size, other.Y_size);
157       std::swap(Y_capacity, other.Y_capacity);
158     }
159
160     const_iterator cbegin() const {
161       return (size() != 0) ? const_iterator(this, 0) : const_iterator
              (this);
162     }
163
164     const_iterator begin() const {
165       return cbegin();
166     }
```

```
167
168     iterator begin() {
169       return (size() ≠ 0) ? iterator(this, 0) : iterator(this);
170     }
171
172     const_iterator cend() const {
173       return const_iterator(this);
174     }
175
176     const_iterator end() const {
177       return cend();
178     }
179
180     iterator end() {
181       return iterator(this);
182     }
183
184     allocator_type get_allocator() const {
185       return allocator;
186     }
187
188     size_type size() const {
189       return std::max(X_size, Y_size);
190     }
191
192     void resize(size_type n) {
193       if (size() > n) {
194         while (size() > n) {
195           pop_back();
196         }
197         return;
198       }
199       append_n(n - size(), V());
200     }
201
202     size_type capacity() const {
203       if (Y_size ≠ 0) {
204         return Y_capacity;
205       }
206       return X_capacity;
207     }
208
209     void reserve(size_type n) {
210       if (n ≤ capacity()) {
211         return;
212       }
213       V* tmp = allocator.allocate(n);
214       std::uninitialized_copy(std::make_move_iterator(X), std::
              make_move_iterator(X + X_size), tmp);
215       if (Y ≠ nullptr) {
216         std::uninitialized_copy(std::make_move_iterator(Y + X_size),
              std::make_move_iterator(Y + Y_size), tmp + X_size);
```

```
217       }
218       allocator.deallocate(X, X_capacity);
219       if (Y ≠ nullptr) {
220         allocator.deallocate(Y, Y_capacity);
221       }
222       X_size = size();
223       X_capacity = n;
224       X = tmp;
225       Y = nullptr;
226       Y_size = 0;
227       Y_capacity = 0;
228     }
229
230     const_reference operator[](size_type i) const {
231       return *index_to_address(i);
232     }
233
234     reference operator[](size_type i) {
235       return *index_to_address(i);
236     }
237
238     const_reference front() const {
239       return *index_to_address(0);
240     }
241
242     reference front() {
243       return *index_to_address(0);
244     }
245
246     const_reference back() const {
247       return *index_to_address(size() - 1);
248     }
249
250     reference back() {
251       return *index_to_address(size() - 1);
252     }
253
254     template <typename... Args>
255     void emplace_back(Args&&... args) {
256       if (Y_size == 0 and X_size < X_capacity) {
257         new (X + X_size) V(std::forward<Args>(args)...);
258         ++X_size;
259         return;
260       }
261       if (Y_size == 0 and X_size == X_capacity) {
262         Y_capacity = 2 * X_capacity;
263         Y = allocator.allocate(Y_capacity);
264         Y_size = X_size;
265       }
266       --X_size;
267       allocator.construct(Y + X_size, std::move(*(X + X_size)));
268       allocator.destroy(X + X_size);
```

```
269       new (Y + Y_size) V(std::forward<Args>(args)...);
270       ++Y_size;
271       if (X_size == 0) {
272         allocator.deallocate(X, X_capacity);
273         X = Y;
274         X_size = Y_size;
275         X_capacity = Y_capacity;
276         Y = nullptr;
277         Y_size = 0;
278         Y_capacity = 0;
279       }
280     }
281
282     void push_back(V const& x) {
283       if (Y_size == 0 and X_size < X_capacity) {
284         allocator.construct(X + X_size, x);
285         ++X_size;
286         return;
287       }
288       if (Y_size == 0 and X_size == X_capacity) {
289         Y_capacity = 2 * X_capacity;
290         Y = allocator.allocate(Y_capacity);
291         Y_size = X_size;
292       }
293       --X_size;
294       allocator.construct(Y + X_size, std::move(*(X + X_size)));
295       allocator.destroy(X + X_size);
296       allocator.construct(Y + Y_size, x);
297       ++Y_size;
298       if (X_size == 0) {
299         allocator.deallocate(X, X_capacity);
300         X = Y;
301         X_size = Y_size;
302         X_capacity = Y_capacity;
303         Y = nullptr;
304         Y_size = 0;
305         Y_capacity = 0;
306       }
307     }
308
309     void push_back(V&& x) {
310       if (Y_size == 0 and X_size < X_capacity) {
311         allocator.construct(X + X_size, std::move(x));
312         ++X_size;
313         return;
314       }
315       if (Y_size == 0 and X_size == X_capacity) {
316         Y_capacity = 2 * X_capacity;
317         Y = allocator.allocate(Y_capacity);
318         Y_size = X_size;
319       }
320       --X_size;
```

```
321       allocator.construct(Y + X_size, std::move(*(X + X_size)));
322       allocator.destroy(X + X_size);
323       allocator.construct(Y + Y_size, std::move(x));
324       ++Y_size;
325       if (X_size == 0) {
326         allocator.deallocate(X, X_capacity);
327         X = Y;
328         X_size = Y_size;
329         X_capacity = Y_capacity;
330         Y = nullptr;
331         Y_size = 0;
332         Y_capacity = 0;
333       }
334     }
335
336     void pop_back() {
337       assert(size() > 0);
338       if (Y_size == 0 and 4 * X_size > X_capacity) {
339         --X_size;
340         allocator.destroy(X + X_size);
341         return;
342       }
343       if (Y_size == 0 and 4 * X_size ≤ X_capacity) {
344         assert(Y_capacity == 0);
345         Y_capacity = 2 * X_size;
346         Y = allocator.allocate(Y_capacity);
347         Y_size = X_size;
348       }
349       --X_size;
350       allocator.construct(Y + X_size, std::move(*(X + X_size)));
351       allocator.destroy(X + X_size);
352       if (X_size ≠ 0) {
353         --X_size;
354         allocator.construct(Y + X_size, std::move(*(X + X_size)));
355         allocator.destroy(X + X_size);
356       }
357       --Y_size;
358       allocator.destroy(Y + Y_size);
359       if (X_size == 0) {
360         allocator.deallocate(X, X_capacity);
361         X = Y;
362         X_size = Y_size;
363         X_capacity = Y_capacity;
364         Y = nullptr;
365         Y_size = 0;
366         Y_capacity = 0;
367       }
368     }
369
370     void clear() {
371       for (size_type i = 0; i ≠ X_size; ++i) {
372         allocator.destroy(X + i);
```

```
373       }
374       for (size_type j = 0; j != Y_size; ++j) {
375         allocator.destroy(Y + j);
376       }
377       if (X != nullptr) {
378         allocator.deallocate(X, X_capacity);
379       }
380       if (Y != nullptr) {
381         allocator.deallocate(Y, Y_capacity);
382       }
383       X = allocator.allocate(1);
384       Y = nullptr;
385       X_size = 0;
386       Y_size = 0;
387       X_capacity = 1;
388       Y_capacity = 0;
389     }
390   };
391 }
392
393 #endif
```

pile.h++

```
1  /*
2    This is an experimental implementation of a levelwise-allocated
3    pile; each level stores a fixed-capacity segment and the directory
4    is a (worst-case-efficient) resizable array.
5
6    Author: Jyrki Katajainen © 2012, 2013, 2015, 2016
7  */
8
9  #ifndef __CPHSTL_PILE__
10 #define __CPHSTL_PILE__
11
12 #include <algorithm> // std::min
13 #include <cassert> // assert macro
14 #include <cstddef> // std::size_t, std::ptrdiff_t
15 #include <initializer_list> // std::initializer_list
16 #include <memory> // std::allocator, std::uninitialized_copy
17 #include "rank-iterator.h++" // cphstl::rank_iterator
18 #include "resizable_array.h++" // cphstl::resizable_array
19 #include <utility> // std::forward, std::move, std::swap
20
21 #ifndef __POPULATION_COUNT__
22 #define __POPULATION_COUNT__
23
24 unsigned int population_count(unsigned int j) {
25   return __builtin_popcount(j);
26 }
27
28 unsigned long population_count(unsigned long j) {
```

```
29   return __builtin_popcountl(j);
30 }
31
32 unsigned long long population_count(unsigned long long j) {
33   return __builtin_popcountll(j);
34 }
35
36 #endif
37
38 #ifndef __WHOLE_NUMBER_LOGARITHM__
39 #define __WHOLE_NUMBER_LOGARITHM__
40
41 std::size_t whole_number_logarithm(std::size_t x) {
42   asm("bsr %0, %0\n"
43       : "=r"(x)
44       : "0" (x)
45   );
46   return x;
47 }
48
49 #endif
50
51 namespace cphstl {
52
53   template <typename V, typename A = std::allocator<V>>
54   class pile {
55   public:
56
57     using value_type = V;
58     using allocator_type = typename A::template rebind<V>::other;
59     using reference = V&;
60     using const_reference = V const&;
61     using pointer = V*;
62     using const_pointer = V const*;
63     using difference_type = std::ptrdiff_t;
64     using size_type = std::size_t;
65     using self_type = pile<V, allocator_type>;
66     using iterator = cphstl::rank_iterator<self_type>;
67     using const_iterator = cphstl::rank_iterator<self_type const>;
68
69   private:
70
71     V* data() noexcept;
72     V const* data() const noexcept;
73
74   protected:
75
76     using other_allocator_type = typename A::template rebind<V*>::
           other;
77     using directory_type = cphstl::resizable_array<V*,
           other_allocator_type>;
78
```

```
79      allocator_type allocator;
80      directory_type directory;
81      size_type n;
82
83      V* index_to_address(size_type rank) const {
84        if (rank < 2) {
85          return directory[0] + rank;
86        }
87        std::size_t h = whole_number_logarithm(rank);
88        return directory[h] + rank - (1 << h);
89      }
90
91      template <typename I>
92      void append(I first, I past) {
93        I p = first;
94        I q = first;
95        try {
96          for ( ; q != past; ++q) {
97            push_back(*q);
98          }
99        }
100       catch (...) {
101         for ( ; p != q; ++p) {
102           pop_back();
103         }
104         throw;
105       }
106     }
107
108     void append_n(size_type k, V const& value) {
109       size_type i = 0;
110       size_type j = 0;
111       try {
112         for ( ; j != k; ++j) {
113           push_back(value);
114         }
115       }
116       catch (...) {
117         for ( ; i != j; ++i) {
118           pop_back();
119         }
120       }
121     }
122
123   public:
124
125     explicit pile(A const& a = A())
126       : allocator(a), directory(a), n(0) {
127       V* p = allocator.allocate(2);
128       directory.push_back(p);
129     }
130
```

```
131     pile(std::initializer_list<V> s)
132       : pile(A()) {
133       append(s.begin(), s.end());
134     }
135
136     pile(size_type n, A const& a)
137       : pile(a) {
138       append_n(n, V());
139     }
140
141     pile(pile const& other)
142       : allocator(other.get_allocator()), directory(other.
              get_allocator()),
143         n(other.size()) {
144       size_type length = 2;
145       size_type i = 0;
146       do {
147         V* s = allocator.allocate(length);
148         size_type j = std::min(i + length, n);
149         std::uninitialized_copy(other.begin() + i, other.begin() + j,
                  s);
150         directory.push_back(s);
151         length = 1 << directory.size();
152         i = j;
153       } while (i != n);
154     }
155
156     template <typename S>
157     pile(S const& other)
158       : allocator(other.get_allocator()), directory(other.
              get_allocator()),
159         n(other.size()) {
160       size_type length = 2;
161       size_type i = 0;
162       do {
163         V* s = allocator.allocate(length);
164         size_type j = std::min(i + length, n);
165         std::uninitialized_copy(other.begin() + i, other.begin() + j,
                  s);
166         directory.push_back(s);
167         length = 1 << directory.size();
168         i = j;
169       } while (i != n);
170     }
171
172     pile(pile&& other)
173       : allocator(other.get_allocator()), directory(other.
              get_allocator()), n(0) {
174       swap(other);
175     }
176
177     ~pile() {
```

```
178       clear();
179       directory.clear();
180     }
181
182     pile& operator=(pile const& other) {
183       if (this ≠ &other) {
184         pile tmp(other);
185         swap(tmp);
186       }
187       return *this;
188     }
189
190     pile& operator=(pile&& other) {
191       if (this ≠ &other) {
192         pile tmp(std::move(other));
193         swap(tmp);
194       }
195       return *this;
196     }
197
198     const_iterator cbegin() const {
199       return (size() ≠ 0) ? const_iterator(this, 0) : const_iterator
              (this);
200     }
201
202     const_iterator begin() const {
203       return cbegin();
204     }
205
206     iterator begin() {
207       return (size() ≠ 0) ? iterator(this, 0) : iterator(this);
208     }
209
210     const_iterator cend() const {
211       return const_iterator(this);
212     }
213
214     const_iterator end() const {
215       return cend();
216     }
217
218     iterator end() {
219       return iterator(this);
220     }
221
222     allocator_type get_allocator() const {
223       return allocator;
224     }
225
226     size_type size() const {
227       return n;
228     }
```

```
229
230     size_type capacity() const {
231       return 1 << directory.size();
232     }
233
234     void resize(size_type new_size) {
235       if (n > new_size) {
236         while (n > new_size) {
237           pop_back();
238         }
239         return;
240       }
241       append_n(new_size - n, V());
242     }
243
244     void reserve(size_type) {
245       // do nothing
246     }
247
248     const_reference operator[](size_type i) const {
249       return *index_to_address(i);
250     }
251
252     reference operator[](size_type i) {
253       return *index_to_address(i);
254     }
255
256     const_reference front() const {
257       return *index_to_address(0);
258     }
259
260     reference front() {
261       return *index_to_address(0);
262     }
263
264     const_reference back() const {
265       return *index_to_address(size() - 1);
266     }
267
268     reference back() {
269       return *index_to_address(size() - 1);
270     }
271
272     template <typename... Args>
273     void emplace_back(Args&&... args) {
274       if (n > 1 and population_count(n) == 1) {
275         size_type h = directory.size();
276         V* p = allocator.allocate(1 << h);
277         directory.push_back(p);
278       }
279       new (index_to_address(n)) V(std::forward<Args>(args)...);
280       n += 1;
```

```
281     }
282
283     void push_back(V const& x) {
284       if (n > 1 and population_count(n) == 1) {
285         size_type h = directory.size();
286         V* p = allocator.allocate(1 << h);
287         directory.push_back(p);
288       }
289       allocator.construct(index_to_address(n), x);
290       n += 1;
291     }
292
293     void push_back(V&& x) {
294       if (n > 1 and population_count(n) == 1) {
295         size_type h = directory.size();
296         V* p = allocator.allocate(1 << h);
297         directory.push_back(p);
298       }
299       allocator.construct(index_to_address(n), std::move(x));
300       n += 1;
301     }
302
303     void pop_back() {
304       assert(n > 0);
305       n -= 1;
306       allocator.destroy(index_to_address(n));
307       size_type h = directory.size();
308       if (h != 1 and population_count(n) == 1) {
309         h -= 1;
310         allocator.deallocate(directory[h], 1 << h);
311         directory.pop_back();
312       }
313     }
314
315     void clear() {
316       while (size() > 0) {
317         pop_back();
318       }
319     }
320
321     void swap(pile& other) {
322       std::swap(allocator, other.allocator);
323       directory.swap(other.directory);
324       std::swap(n, other.n);
325     }
326   };
327 }
328
329 #endif
```

sliced_array.h++

```
/*
  This is an experimental implementation of a page-table-like array
  often used in the implementation of the standard-library deque;
  each page is a fixed-capacity array and the directory is a
  (worst-case-efficient) resizable array.

  Author: Jyrki Katajainen @ 2012, 2013, 2015, 2016
*/

#ifndef __CPHSTL_SLICED_ARRAY__
#define __CPHSTL_SLICED_ARRAY__

#include <cassert> // assert macro
#include <cstddef> // std::size_t, std::ptrdiff_t
#include <initializer_list> // std::initializer_list
#include <memory> // std::allocator, std::uninitialized_copy
#include "rank-iterator.h++" // cphstl::rank_iterator
#include "resizable_array.h++" // cphstl::resizable_array
#include <utility> // std::forward, std::move, std::swap

namespace cphstl {

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

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

  private:

    V* data() noexcept;
    V const* data() const noexcept;

  protected:

    size_type const slice_size = 512;
    size_type const shift_amount = 9;
    size_type const mask = 511;
    using B = typename A::template rebind<V*>::other;
    using directory_type = cphstl::resizable_array<V*, B>;

    allocator_type allocator;
```

```
53    B pointer_allocator;
54    directory_type directory;
55    size_type n;
56
57    V* index_to_address(size_type rank) const {
58      return directory[rank >> shift_amount] + (rank bitand mask);
59    }
60
61    template <typename I>
62    void append(I first, I past) {
63      I p = first;
64      I q = first;
65      try {
66        for ( ; q =/= past; ++q) {
67          push_back(*q);
68        }
69      }
70      catch (...) {
71        for ( ; p =/= q; ++p) {
72          pop_back();
73        }
74        throw;
75      }
76    }
77
78    void append_n(size_type k, V const& value) {
79      size_type i = 0;
80      size_type j = 0;
81      try {
82        for ( ; j =/= k; ++j) {
83          push_back(value);
84        }
85      }
86      catch (...) {
87        for ( ; i =/= j; ++i) {
88          pop_back();
89        }
90      }
91    }
92
93  public:
94
95    explicit sliced_array(A const& a = A())
96      : allocator(a), pointer_allocator(B(a)), directory(B(a)), n(0)
          {
97    }
98
99    sliced_array(std::initializer_list<V> s)
100     : sliced_array(A()) {
101     append(s.begin(), s.end());
102   }
103
```

```
104     sliced_array(size_type n, A const& a)
105       : sliced_array(a) {
106       append_n(n, V());
107     }
108
109     sliced_array(sliced_array const& other)
110       : allocator(other.allocator), pointer_allocator(other.
              pointer_allocator),
111         directory(other.pointer_allocator), n(0) {
112       directory_type tmp;
113       for (size_type i = 0; i < other.directory.size(); ++i) {
114         V* p = allocator.allocate(slice_size);
115         tmp.push_back(p);
116         std::uninitialized_copy(other.directory[i], other.directory[i
                  ] + slice_size, p);
117       }
118       directory = std::move(tmp);
119       n = other.size();
120     }
121
122     sliced_array(sliced_array&& other)
123       : allocator(other.allocator), pointer_allocator(other.
              pointer_allocator),
124         directory(other.pointer_allocator), n(0) {
125       swap(other);
126     }
127
128     ~sliced_array() {
129       clear();
130     }
131
132     sliced_array& operator=(sliced_array const& other) {
133       if (this != &other) {
134         sliced_array tmp(other);
135         swap(tmp);
136       }
137       return *this;
138     }
139
140     sliced_array& operator=(sliced_array&& other) {
141       if (this != &other) {
142         sliced_array tmp(std::move(other));
143         swap(tmp);
144       }
145       return *this;
146     }
147
148     void swap(sliced_array& other) {
149       std::swap(allocator, other.allocator);
150       std::swap(pointer_allocator, other.pointer_allocator);
151       directory.swap(other.directory);
152       std::swap(n, other.n);
```

```
153     }
154
155     const_iterator cbegin() const {
156       return (size() != 0) ? const_iterator(this, 0) : const_iterator
              (this);
157     }
158
159     const_iterator begin() const {
160       return cbegin();
161     }
162
163     iterator begin() {
164       return (size() != 0) ? iterator(this, 0) : iterator(this);
165     }
166
167     const_iterator cend() const {
168       return const_iterator(this);
169     }
170
171     const_iterator end() const {
172       return cend();
173     }
174
175     iterator end() {
176       return iterator(this);
177     }
178
179     allocator_type get_allocator() const {
180       return allocator;
181     }
182
183     size_type size() const {
184       return n;
185     }
186
187     void resize(size_type new_size) {
188       if (n > new_size) {
189         while (n > new_size) {
190           pop_back();
191         }
192         return;
193       }
194       append_n(new_size - n, V());
195     }
196
197     size_type capacity() const {
198       return size_type(-1);
199     }
200
201     void reserve(size_type) {
202     }
203
```

```
204     const_reference operator[](size_type i) const {
205       return *index_to_address(i);
206     }
207
208     reference operator[](size_type i) {
209       return *index_to_address(i);
210     }
211
212     const_reference front() const {
213       return *index_to_address(0);
214     }
215
216     reference front() {
217       return *index_to_address(0);
218     }
219
220     const_reference back() const {
221       return *index_to_address(size() - 1);
222     }
223
224     reference back() {
225       return *index_to_address(size() - 1);
226     }
227
228     template <typename... Args>
229     void emplace_back(Args&&... args) {
230       size_type offset = n bitand mask;
231       if (offset == 0) {
232         V* p = allocator.allocate(slice_size);
233         directory.push_back(p);
234       }
235       new (index_to_address(n)) V(std::forward<Args>(args)...);
236       n += 1;
237     }
238
239     void push_back(V const& x) {
240       size_type offset = n bitand mask;
241       if (offset == 0) {
242         V* p = allocator.allocate(slice_size);
243         directory.push_back(p);
244       }
245       allocator.construct(directory[n >> shift_amount] + offset, x);
246       n += 1;
247     }
248
249     void push_back(V&& x) {
250       size_type offset = n bitand mask;
251       if (offset == 0) {
252         V* p = allocator.allocate(slice_size);
253         directory.push_back(p);
254       }
255       allocator.construct(directory[n >> shift_amount] + offset, std
```

```
            ::move(x));
256       n += 1;
257     }
258
259     void pop_back() {
260       assert(n > 0);
261       n -= 1;
262       V* base = directory[n >> shift_amount];
263       size_type offset = n bitand mask;
264       allocator.destroy(base + offset);
265       if (offset == 0) {
266         allocator.deallocate(base, slice_size);
267         directory.pop_back();
268       }
269     }
270
271     void clear() {
272       while (n > 0) {
273         pop_back();
274       }
275     }
276   };
277 }
278
279 #endif
```

fixed_capacity_hat.h++

```
1  /*
2    This is an experimental implementation of a hashed array tree which
3    has a fixed capacity. The capacity must be set using the reserve()
4    function and it can only be set once.
5
6    Author: Jyrki Katajainen © 2012, 2013, 2016
7  */
8
9  #include <algorithm> // std::max
10 #include <cassert> // assert macro
11 #include <cmath> // ilogb
12 #include "contiguous_array.h++"
13 #include <cstddef> // std::size_t, std::ptrdiff_t
14 #include "rank-iterator.h++"
15 #include <utility> // std::forward, std::move, std::swap
16
17 namespace cphstl {
18
19   template <typename H, typename B>
20   class space_efficient_array;
21
22   template <typename V, typename A = std::allocator<V>>
23   class fixed_capacity_hat {
24   public:
```

```
25
26    using value_type = V;
27    using allocator_type = typename A::template rebind<V>::other;
28    using reference = V&;
29    using const_reference = V const&;
30    using pointer = V*;
31    using const_pointer = V const*;
32    using difference_type = std::ptrdiff_t;
33    using size_type = std::size_t;
34    using self_type = fixed_capacity_hat<V, allocator_type>;
35    using iterator = cphstl::rank_iterator<self_type>;
36    using const_iterator = cphstl::rank_iterator<self_type const>;
37
38  protected:
39
40    template <typename H, typename B>
41    friend class space_efficient_array;
42
43    using other_allocator_type = typename A::template rebind<V*>::
          other;
44    using directory_type = cphstl::contiguous_array<V*,
          other_allocator_type>;
45
46    allocator_type allocator;
47    directory_type directory;
48    size_type n;
49    size_type m; // a power of two
50    size_type shift_amount; // lg m
51    size_type mask;
52
53    V* index_to_address(size_type rank) const {
54      return directory[rank >> shift_amount] + (rank bitand mask);
55    }
56
57    void append_n(size_type k, V const& value) {
58      size_type i = 0;
59      size_type j = 0;
60      try {
61        for ( ; j != k; ++j) {
62          push_back(value);
63        }
64      }
65      catch (...) {
66        for ( ; i != j; ++i) {
67          pop_back();
68        }
69      }
70    }
71
72  public:
73
74    explicit fixed_capacity_hat(A const& a = A())
```

```
75      : allocator(a), directory(a), n(0), m(0), shift_amount(0), mask
            (0) {
76    }
77
78    fixed_capacity_hat(fixed_capacity_hat const& other)
79      : allocator(other.allocator), directory(other.allocator), n(0),
80        m(0), shift_amount(0), mask(0) {
81      directory_type tmp;
82      tmp.reserve(other.m);
83      for (size_type i = 0; i != other.directory.size(); ++i) {
84        V* p = allocator.allocate(m);
85        tmp.push_back(p);
86        std::uninitialized_copy(other.directory[i], other.directory[i
              ] + m, p);
87      }
88      directory = std::move(tmp);
89      n = other.n;
90      m = other.m;
91      shift_amount = other.shift_amount;
92      mask = other.mask;
93    }
94
95    fixed_capacity_hat(fixed_capacity_hat&& other)
96      : allocator(other.allocator), directory(other.allocator), n(0),
97        m(0), shift_amount(0), mask(0) {
98      swap(other);
99    }
100
101   ~fixed_capacity_hat() {
102     clear();
103   }
104
105   fixed_capacity_hat& operator=(fixed_capacity_hat const& other)
          {
106     if (this != &other) {
107       fixed_capacity_hat tmp(other);
108       swap(tmp);
109     }
110     return *this;
111   }
112
113   fixed_capacity_hat& operator=(fixed_capacity_hat&& other) {
114     if (this != &other) {
115       fixed_capacity_hat tmp(std::move(other));
116       swap(tmp);
117     }
118     return *this;
119   }
120
121   void swap(fixed_capacity_hat& other) {
122     std::swap(allocator, other.allocator);
123     directory.swap(other.directory);
```

```
124       std::swap(n, other.n);
125       std::swap(m, other.m);
126       std::swap(shift_amount, other.shift_amount);
127       std::swap(mask, other.mask);
128     }
129
130     const_iterator cbegin() const {
131       return (size() != 0) ? const_iterator(this, 0) : const_iterator
              (this);
132     }
133
134     const_iterator begin() const {
135       return cbegin();
136     }
137
138     iterator begin() {
139       return (size() != 0) ? iterator(this, 0) : iterator(this);
140     }
141
142     const_iterator cend() const {
143       return const_iterator(this);
144     }
145
146     const_iterator end() const {
147       return cend();
148     }
149
150     iterator end() {
151       return iterator(this);
152     }
153
154     allocator_type get_allocator() const {
155       return allocator;
156     }
157
158     size_type size() const {
159       return n;
160     }
161
162     void resize(size_type new_size) {
163       if (n > new_size) {
164         while (n > new_size) {
165           pop_back();
166         }
167         return;
168       }
169       append_n(new_size - n, V());
170       n = new_size;
171     }
172
173     size_type capacity() const {
174       return m * m;
```

```
175     }
176
177     void reserve(size_type n) {
178       assert(m == 0);
179       size_type lg_n = ilogb(std::max(size_type(1), n));
180       size_type new_m = 1 << (lg_n / 2); // about sqrt of n
181       if (new_m * new_m < n) {
182         new_m = 2 * new_m;
183       }
184       directory.reserve(new_m);
185       m = new_m;
186       shift_amount = ilogb(m);
187       mask = m - 1;
188     }
189
190     const_reference operator[](size_type i) const {
191       return *index_to_address(i);
192     }
193
194     reference operator[](size_type i) {
195       return *index_to_address(i);
196     }
197
198     template <typename... Args>
199     void emplace_back(Args&&... args) {
200       assert(n < capacity());
201       size_type offset = n bitand mask;
202       if (offset == 0) {
203         V* p = allocator.allocate(m);
204         directory.push_back(p);
205       }
206       new (index_to_address(n)) V(std::forward<Args>(args)...);
207       n += 1;
208     }
209
210     void push_back(V const& x) {
211       assert(n < capacity());
212       size_type offset = n bitand mask;
213       if (offset == 0) {
214         V* p = allocator.allocate(m);
215         directory.push_back(p);
216       }
217       allocator.construct(directory[n >> shift_amount] + offset, x);
218       n += 1;
219     }
220
221     void push_back(V&& x) {
222       assert(n < capacity());
223       size_type offset = n bitand mask;
224       if (offset == 0) {
225         V* p = allocator.allocate(m);
226         directory.push_back(p);
```

```
227       }
228       allocator.construct(directory[n >> shift_amount] + offset, std
              ::move(x));
229       n += 1;
230     }
231
232     void pop_back() {
233       assert(n != 0);
234       n -= 1;
235       V* base = directory[n >> shift_amount];
236       size_type offset = n bitand mask;
237       allocator.destroy(base + offset);
238       if (offset == 0) {
239         allocator.deallocate(base, m);
240         directory.pop_back();
241       }
242     }
243
244     void clear() {
245       while (n > 0) {
246         pop_back();
247       }
248       while (directory.size() > 0) {
249         directory.pop_back();
250       }
251       m  = 0;
252       shift_amount = 0;
253       mask = 0;
254     }
255   };
256 }
```

space_efficient_array.h++

```
1  /*
2    This is an experimental implementation of a space-efficient array
3    that consists of a pool of fixed-capacity hashed array trees; the
4    directory is a (worst-case-efficient) resizable array.
5
6    Author: Jyrki Katajainen @ 2012, 2016
7  */
8
9  #ifndef __CPHSTL_SPACE_EFFICIENT_ARRAY__
10 #define __CPHSTL_SPACE_EFFICIENT_ARRAY__
11
12 #include <cassert> // assert macro
13 #include <cstddef> // std::size_t, std::ptrdiff_t
14 #include "fixed_capacity_hat.h++" // cphstl::fixed_capacity_hat
15 #include <initializer_list> // std::initializer_list
16 #include <memory> // std::allocator
17 #include "rank-iterator.h++" // cphstl::rank_iterator
18 #include "resizable_array.h++" // cphstl::resizable_array
```

```
#include <utility> // std::forward, std::move, std::swap

#ifndef __POPULATION_COUNT__
#define __POPULATION_COUNT__

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);
}

#endif

#ifndef __WHOLE_NUMBER_LOGARITHM__
#define __WHOLE_NUMBER_LOGARITHM__

std::size_t whole_number_logarithm(std::size_t x) {
  asm("bsr %0, %0\n"
      : "=r"(x)
      : "0" (x)
  );
  return x;
}

#endif

namespace cphstl {

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

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

  private:

```

```
71    V* data() noexcept;
72    V const* data() const noexcept;
73
74  protected:
75
76    using H = cphstl::fixed_capacity_hat<V, allocator_type>;
77    using B = typename A::template rebind<H>::other;
78    using directory_type = cphstl::resizable_array<H, B>;
79
80    allocator_type allocator;
81    directory_type directory;
82    size_type n;
83
84    V* index_to_address(size_type rank) const {
85      if (rank < 2) {
86        return directory[0].index_to_address(rank);
87      }
88      std::size_t h = whole_number_logarithm(rank);
89      size_type offset = rank - (1 << h);
90      return directory[h].index_to_address(offset);
91    }
92
93    template <typename S, typename I>
94    void append(S& sequence, I some, I past) {
95      I p = some;
96      I q = some;
97      try {
98        for ( ; q != past; ++q) {
99          sequence.push_back(*q);
100       }
101     }
102     catch (...) {
103       for ( ; p != q; ++p) {
104         sequence.pop_back();
105       }
106       throw;
107     }
108   }
109
110   void append_n(size_type k, V const& value) {
111     size_type i = 0;
112     size_type j = 0;
113     try {
114       for ( ; j != k; ++j) {
115         push_back(value);
116       }
117     }
118     catch (...) {
119       for ( ; i != j; ++i) {
120         pop_back();
121       }
122     }
```

```
123     }
124
125   public:
126
127     explicit space_efficient_array(A const& a = A())
128       : allocator(a), directory(a), n(0) {
129       directory.push_back(std::move(H(allocator)));
130       directory[0].reserve(2);
131     }
132
133     space_efficient_array(std::initializer_list<V> s)
134       : space_efficient_array(A()) {
135       append(*this, s.begin(), s.end());
136     }
137
138     space_efficient_array(space_efficient_array const& other)
139       : allocator(other.allocator), directory(other.allocator), n(0)
            {
140       space_efficient_array tmp;
141       append(tmp, other.begin(), other.end());
142       (*this).swap(tmp);
143     }
144
145     space_efficient_array(space_efficient_array&& other)
146       : allocator(other.allocator), directory(other.allocator), n(0)
            {
147       (*this).swap(other);
148     }
149
150     ~space_efficient_array() {
151       clear();
152       directory.clear();
153     }
154
155     space_efficient_array& operator=(space_efficient_array const&
          other) {
156       if (this != &other) {
157         space_efficient_array tmp(other);
158         (*this).swap(tmp);
159       }
160       return *this;
161     }
162
163     space_efficient_array& operator=(space_efficient_array&& other)
          {
164       if (this != &other) {
165         space_efficient_array tmp(std::move(other));
166         (*this).swap(tmp);
167       }
168       return *this;
169     }
170
```

```
171     void swap(space_efficient_array& other) {
172       std::swap(allocator, other.allocator);
173       directory.swap(other.directory);
174       std::swap(n, other.n);
175     }
176
177     const_iterator cbegin() const {
178       return (size() != 0) ? const_iterator(this, 0) : const_iterator
              (this);
179     }
180
181     const_iterator begin() const {
182       return cbegin();
183     }
184
185     iterator begin() {
186       return (size() != 0) ? iterator(this, 0) : iterator(this);
187     }
188
189     const_iterator cend() const {
190       return const_iterator(this);
191     }
192
193     const_iterator end() const {
194       return cend();
195     }
196
197     iterator end() {
198       return iterator(this);
199     }
200
201     allocator_type get_allocator() const {
202       return allocator;
203     }
204
205     size_type size() const {
206       return n;
207     }
208
209     void resize(size_type new_size) {
210       if (n > new_size) {
211         while (n > new_size) {
212           pop_back();
213         }
214         return;
215       }
216       append_n(new_size - n, V());
217     }
218
219     size_type capacity() const {
220       return size_type(-1);
221     }
```

```
222
223     void reserve(size_type) {
224     }
225
226     const_reference operator[](size_type i) const {
227       return *index_to_address(i);
228     }
229
230     reference operator[](size_type i) {
231       return *index_to_address(i);
232     }
233
234     const_reference front() const {
235       return *index_to_address(0);
236     }
237
238     reference front() {
239       return *index_to_address(0);
240     }
241
242     const_reference back() const {
243       return *index_to_address(size() - 1);
244     }
245
246     reference back() {
247       return *index_to_address(size() - 1);
248     }
249
250     template <typename... Args>
251     void emplace_back(Args&&... args) {
252       size_type h = directory.size();
253       if (n > 1 and population_count(n) == 1) {
254         directory.push_back(std::move(H(allocator)));
255         directory[h].reserve(1 << h);
256         h += 1;
257       }
258       directory[h - 1].emplace_back(std::forward<Args>(args)...);
259       n += 1;
260     }
261
262     void push_back(V const& x) {
263       size_type h = directory.size();
264       if (n > 1 and population_count(n) == 1) {
265         directory.push_back(std::move(H(allocator)));
266         directory[h].reserve(1 << h);
267         h += 1;
268       }
269       directory[h - 1].push_back(x);
270       n += 1;
271     }
272
273     void push_back(V&& x) {
```

```
274       size_type h = directory.size();
275       if (n > 1 and population_count(n) == 1) {
276         directory.push_back(std::move(H(allocator)));
277         directory[h].reserve(1 << h);
278         h += 1;
279       }
280       directory[h - 1].push_back(std::move(x));
281       n += 1;
282     }
283
284     void pop_back() {
285       assert(n != 0);
286       n -= 1;
287       size_type h = directory.size();
288       directory[h - 1].pop_back();
289       if (h != 1 and population_count(n) == 1) {
290         directory[h - 1].clear();
291         directory.pop_back();
292       }
293     }
294
295     void clear() {
296       while (size() > 0) {
297         pop_back();
298       }
299     }
300   };
301 }
302
303 #endif
```

Containers

leda-array.h++

```
1  /*
2    The interface of cphleda::array
3
4    Authors: Jyrki Katajainen, Bo Simonsen © 2009, 2016
5  */
6
7  #include <algorithm> // std::unique, std::sort, std::random_shuffle,
       std::lower_bound, std::upper_bound
8  #include <cassert> // assert macro
9  #include "leda-compare-functions.i++"
10 #include <memory> // std::allocator
11 #include <vector> // std::vector
12 #include <sstream> // std::stringstream
13 #include <utility> // std::swap, std::pair
14
15 namespace cphleda {
```

```
16
17  template <typename V,  typename K = std::vector<V, std::allocator<V
        >>>
18  class array {
19  public:
20
21    using value_type = V;
22    using kernel_type = K;
23    using iterator = typename K::iterator;
24    using const_iterator = typename K::const_iterator;
25    using item = iterator;
26    using size_type = int;
27
28    array(size_type, size_type);
29    array(size_type);
30    array();
31    array(size_type, V, V);
32    array(size_type, V, V, V);
33    array(size_type, V, V, V, V);
34
35    array(array const&);
36    array& operator=(array const&);
37
38    item first_item();
39    item last_item();
40    item next_item(item);
41    item prev_item(item);
42
43    V& inf(item);
44
45    iterator begin();
46    iterator end();
47    const_iterator begin() const;
48    const_iterator end() const;
49
50    V& get(size_type);
51    V const& get(size_type) const;
52    void set(size_type, V);
53    V& operator[](size_type);
54
55    void copy(size_type, array const&, size_type);
56    void copy(size_type, size_type);
57    void resize(size_type, size_type);
58    void resize(size_type);
59
60    size_type low() const;
61    size_type high() const;
62    size_type size() const;
63
64    void init(V);
65    bool C_style() const;
66
```

```
67      void swap(size_type, size_type);
68
69      void sort(size_type (*)(V const&, V const&));
70      void sort(size_type, size_type);
71      void sort();
72
73      void sort(size_type (*)(V const&, V const&), size_type,
            size_type);
74      void permute();
75      void permute(size_type, size_type);
76
77      size_type binary_search(size_type (*)(V const&, V const&), V);
78      size_type binary_search(V);
79      size_type binary_locate(size_type (*)(V const&, V const&), V);
80      size_type binary_locate(V);
81
82      size_type unique();
83
84      void print(std::ostream&, char = '␣');
85      void print(char = '␣');
86      void print(char const*, char = '␣');
87
88      void read(std::istream&);
89      void read();
90      void read(char const*);
91
92    protected:
93
94      void fill(K&, size_type, size_type);
95
96      template <typename S, typename I>
97      void append(S&, I, I);
98
99      std::pair<iterator, iterator> get_iterators(size_type, size_type)
            ;
100
101   private:
102
103     size_type s;
104     size_type e;
105     kernel_type kernel;
106   };
107
108   template <typename V, typename K>
109   std::istream& operator>>(std::istream&, array<V, K>&);
110
111   template <typename V, typename K>
112   std::ostream& operator<<(std::ostream&, array<V, K>&);
113 }
114
115 #include "leda-array.i++" // implementation
```

leda-array.i++

```
1  /*
2    An implementation of cphleda::array
3
4    Authors: Jyrki Katajainen, Bo Simonsen © 2009, 2016
5  */
6
7  namespace cphleda {
8
9    template <typename V, typename K>
10   array<V, K>::array(size_type a, size_type b)
11     : s(0), e(-1), kernel() {
12     fill(kernel, 0, b - a);
13     s = a;
14     e = b;
15   }
16
17   template <typename V, typename K>
18   array<V, K>::array(size_type n)
19     : s(0), e(-1), kernel() {
20     fill(kernel, 0, n);
21     e = n - 1;
22   }
23
24   template <typename V, typename K>
25   array<V, K>::array()
26     : s(0), e(-1), kernel() {
27   }
28
29   template <typename V, typename K>
30   array<V, K>::array(size_type low, V x, V y)
31     : s(0), e(-1), kernel() {
32     std::initializer_list<V> list = {x, y};
33     append(kernel, list.begin(), list.end());
34     s = low;
35     e = low + 1;
36   }
37
38   template <typename V, typename K>
39   array<V, K>::array(size_type low, V x, V y, V z)
40     : s(0), e(-1), kernel() {
41     std::initializer_list<V> list = {x, y, z};
42     append(kernel, list.begin(), list.end());
43     s = low;
44     e = low + 2;
45   }
46
47   template <typename V, typename K>
48   array<V, K>::array(size_type low, V x, V y, V z, V w)
49     : s(0), e(-1), kernel() {
```

```
50    std::initializer_list<V> list = {x, y, z, w};
51    append(kernel, list.begin(), list.end());
52    s = low;
53    e = low + 3;
54  }
55
56  template <typename V, typename K>
57  array<V, K>::array(array const& a)
58    : s(0), e(-1), kernel() {
59    K tmp;
60    append(tmp, a.begin(), a.end());
61    kernel.swap(tmp);
62    s = a.low();
63    e = a.high();
64  }
65
66  template <typename V, typename K>
67  array<V, K>& array<V, K>::operator=(array const& other) {
68    if (this ≠ &other) {
69      K tmp;
70      append(tmp, other.begin(), other.end());
71      kernel.swap(tmp);
72      s = other.low();
73      e = other.high();
74    }
75    return *this;
76  }
77
78  template <typename V, typename K>
79  V& array<V, K>::get(size_type i) {
80    assert(low() ≤ i and i ≤ high());
81    return kernel[i - low()];
82  }
83
84  template <typename V, typename K>
85  V const& array<V, K>::get(size_type i) const {
86    assert(low() ≤ i and i ≤ high());
87    return kernel[i - low()];
88  }
89
90  template <typename V, typename K>
91  void array<V, K>::set(size_type i, V e) {
92    assert(low() ≤ i and i ≤ high());
93    get(i) = e;
94  }
95
96  template <typename V, typename K>
97  V& array<V, K>::operator[](size_type i) {
98
99  #if not defined(LEDA_CHECKING_OFF)
100
101   if (i < low() or i > high()) {
```

```
102       throw std::out_of_range{"ERROR array: index out of range"};
103     }
104
105 #endif
106
107     return get(i);
108   }
109
110   template <typename V, typename K>
111   void array<V, K>::copy(size_type x, array<V, K> const& other,
          size_type y) {
112     assert(low() ≤ x and x ≤ high());
113     assert(other.low() ≤ y and y ≤ other.high());
114     set(x, other.get(y));
115   }
116
117   template <typename V, typename K>
118   void array<V, K>::copy(size_type x, size_type y) {
119     assert(low() ≤ x and x ≤ high());
120     assert(low() ≤ y and y ≤ high());
121     set(x, get(y));
122   }
123
124   template <typename V, typename K>
125   void array<V, K>::resize(size_type a, size_type b) {
126     assert(a ≤ b);
127     K tmp;
128     if (e < a) {
129       fill(tmp, 0, b - a + 1);
130     }
131     else if (e ≤ b) {
132       if (s < a) {
133         append(tmp, begin() + (a - s), end());
134         fill(tmp, e - a + 1, b - a + 1);
135       }
136       else {
137         fill(tmp, 0, s - a);
138         append(tmp, begin(), end());
139         fill(tmp, tmp.size(), b - a + 1);
140       }
141     }
142     else if (s < a) {
143       append(tmp, kernel.begin() + (a - s), kernel.begin() + (b - s +
              1));
144     }
145     else if (s ≤ b) {
146       fill(tmp, 0, s - a);
147       append(tmp, kernel.begin(), kernel.begin() + (b - s + 1));
148     }
149     else {
150       assert(s > b);
151       fill(tmp, 0, b - a + 1);
```

```
152     }
153     kernel.swap(tmp);
154     s = a;
155     e = b;
156   }
157
158   template <typename V, typename K>
159   void array<V, K>::resize(size_type n) {
160     resize(0, n - 1);
161   }
162
163   template <typename V, typename K>
164   typename array<V, K>::size_type array<V, K>::low() const {
165     return s;
166   }
167
168   template <typename V, typename K>
169   typename array<V, K>::size_type array<V, K>::high() const {
170     return e;
171   }
172
173   template <typename V, typename K>
174   typename array<V, K>::size_type array<V, K>::size() const {
175     return e - s + 1;
176   }
177
178   template <typename V, typename K>
179   void array<V, K>::init(V x) {
180     for(size_type i = low(); i ≤ high(); ++i) {
181       set(i, x);
182     }
183   }
184
185   template <typename V, typename K>
186   bool array<V, K>::C_style() const {
187     return low() == 0;
188   }
189
190   template <typename V, typename K>
191   void array<V, K>::swap(size_type i, size_type j) {
192     std::swap(get(i), get(j));
193   }
194
195   template <typename V, typename K>
196   typename array<V, K>::size_type array<V, K>::unique() {
197     std::pair<iterator, iterator> p = get_iterators(low(), high());
198     iterator new_end = std::unique(p.first, p.second);
199     size_type i = (new_end - p.first) + low();
200     return i - 1;
201   }
202
203   /* sort functions */
```

```
204
205  template <typename V, typename K>
206  void array<V, K>::sort(size_type (*cmp)(V const&, V const&)) {
207    sort(cmp, low(), high());
208  }
209
210  template <typename V, typename K>
211  void array<V, K>::sort(size_type l, size_type h) {
212    sort(compare, l, h);
213  }
214
215  template <typename V, typename K>
216  void array<V, K>::sort() {
217    sort(compare);
218  }
219
220  template <typename V, typename K>
221  void array<V, K>::sort(size_type (*cmp)(V const&, V const&),
         size_type l, size_type h) {
222    std::pair<iterator, iterator> p = get_iterators(l, h);
223    std::sort(p.first, p.second, stl_compare_less<V>(cmp));
224  }
225
226  /* permute functions */
227
228  template <typename V, typename K>
229  void array<V, K>::permute() {
230    permute(low(), high());
231  }
232
233  template <typename V, typename K>
234  void array<V, K>::permute(size_type l, size_type h) {
235    std::pair<iterator, iterator> p = get_iterators(l, h);
236    std::random_shuffle(p.first, p.second);
237  }
238
239  /* binary-search functions */
240
241  template <typename V, typename K>
242  typename array<V, K>::size_type
243  array<V, K>::binary_search(size_type (*cmp)(V const&, V const&),
         V x) {
244    std::pair<iterator, iterator> p = get_iterators(low(), high());
245    iterator it = std::lower_bound(p.first, p.second, x,
           stl_compare_less<V>(cmp));
246    if (it == p.second) {
247      return low() - 1;
248    }
249    return (it - p.first) + low();
250  }
251
252  template <typename V, typename K>
```

```
253  typename array<V, K>::size_type array<V, K>::binary_search(V x) {
254    return binary_search(compare, x);
255  }
256
257  template <typename V, typename K>
258  typename array<V, K>::size_type
259  array<V, K>::binary_locate(size_type (*cmp)(V const&, V const&),
         V x) {
260    std::pair<iterator, iterator> p = get_iterators(low(), high());
261    iterator it = std::upper_bound(p.first, p.second, x,
262                                   stl_compare_less<V>(cmp));
263    if (it == p.second) {
264      return low() - 1;
265    }
266    size_type i = (it - p.first) - 1 + low();
267    if (cmp(x, get(i)) > 0) {
268      return low() - 1;
269    }
270    return i;
271  }
272
273  template <typename V, typename K>
274  typename array<V, K>::size_type array<V, K>::binary_locate(V x) {
275    return binary_locate(compare, x);
276  }
277
278  /* print functions */
279
280  template <typename V, typename K>
281  void array<V, K>::print(std::ostream& o, char space) {
282    std::stringstream sss;
283    sss << space;
284    for(size_type i = low(); i ≤ high(); ++i) {
285      std::stringstream ss;
286      ss << get(i);
287      o << ss.str();
288      if(i ≠ high()) {
289        o << sss.str();
290      }
291    }
292  }
293
294  template <typename V, typename K>
295  void array<V, K>::print(char space) {
296    print(std::cout, space);
297  }
298
299  template <typename V, typename K>
300  void array<V, K>::print(char const* header, char space) {
301    std::cout << header << std::endl;
302    print(std::cout, space);
303  }
```

```
304
305  /* read functions */
306
307  template <typename V, typename K>
308  void array<V, K>::read(std::istream& s) {
309    V v;
310    for(size_type i = low(); i ≤ high(); ++i) {
311      s >> v;
312      set(i, v);
313    }
314  }
315
316  template <typename V, typename K>
317  void array<V, K>::read() {
318    read(std::cin);
319  }
320
321  template <typename V, typename K>
322  void array<V, K>::read(const char* header) {
323    std::cout << header;
324    read();
325  }
326
327  /* helpers */
328
329  template <typename V, typename K>
330  void array<V, K>::fill(K& data, size_type j, size_type n) {
331    size_type k = j;
332    try {
333      for ( ; k ≠ n; ++k) {
334        data.push_back(V());
335      }
336    }
337    catch (...) {
338      for ( ; j ≠ k; ++j) {
339        data.pop_back();
340      }
341      throw;
342    }
343  }
344
345  template <typename V, typename K>
346  template <typename S, typename I>
347  void array<V, K>::append(S& sequence, I some, I past) {
348    I p = some;
349    I q = some;
350    try {
351      for ( ; q ≠ past; ++q) {
352        sequence.push_back(*q);
353      }
354    }
355    catch (...) {
```

```
356       for ( ; p ≠ q; ++p) {
357         sequence.pop_back();
358       }
359       throw;
360     }
361   }
362
363   template <typename V, typename K>
364   std::pair<typename array<V, K>::iterator, typename array<V, K>::
          iterator>
365   array<V, K>::get_iterators(size_type l, size_type h) {
366     size_type start_index = l - low();
367     size_type end_index = h - low();
368     iterator begin_it = kernel.begin();
369     iterator end_it = kernel.begin();
370     begin_it += start_index;
371     end_it += end_index + 1;
372     return std::pair<iterator, iterator>(begin_it, end_it);
373   }
374
375   /* we use iterators to implement next_item, prev_item, and inf */
376
377   template <typename V, typename K>
378   typename array<V, K>::item array<V, K>::first_item() {
379     return kernel.begin();
380   }
381
382   template <typename V, typename K>
383   typename array<V, K>::item array<V, K>::last_item() {
384     return prev_item(kernel.end());
385   }
386
387   template <typename V, typename K>
388   typename array<V, K>::item array<V, K>::next_item(item a) {
389     iterator p(a);
390     ++p;
391     return p;
392   }
393
394   template <typename V, typename K>
395   typename array<V, K>::item array<V, K>::prev_item(item a) {
396     iterator p(a);
397     --p;
398     return p;
399   }
400
401   template <typename V, typename K>
402   V& array<V, K>::inf(item a) {
403     iterator p(a);
404     return *p;
405   }
406
```

```
407   /* iterators */
408
409   template <typename V, typename K>
410   typename array<V, K>::iterator array<V, K>::begin() {
411     return kernel.begin();
412   }
413
414   template <typename V, typename K>
415   typename array<V, K>::iterator array<V, K>::end() {
416     return kernel.end();
417   }
418
419   template <typename V, typename K>
420   typename array<V, K>::const_iterator array<V, K>::begin() const {
421     return kernel.begin();
422   }
423
424   template <typename V, typename K>
425   typename array<V, K>::const_iterator array<V, K>::end() const {
426     return kernel.end();
427   }
428
429   /* operator << and operator >> */
430
431   template <typename V, typename K>
432   std::istream& operator>>(std::istream& in, array<V, K>& a) {
433     a.read(in);
434     return in;
435   }
436
437   template <typename V, typename K>
438   std::ostream& operator<<(std::ostream& out, array<V, K>& a) {
439     a.print(out);
440     return out;
441   }
442 }
```

stl-vector.h++

```
1  /*
2    According to the C++ standard [2014], a vector is a sequence that
3    supports random-access iterators. More precisely, it satisfies all
4    of the requirements of a container, of a reversible container, and
5    of a sequence container, including most of the optional sequence
6    container requirements, of an allocator-aware container, and, for
7    an element type other than bool, of a contiguous container. The
8    exceptions are the push_front, pop_front, and emplace_front member
9    functions, which are not provided.
10
11   Performance Engineering Laboratory © 2008, 2016
12 */
13
```

```
14 #ifndef __CPHSTL_VECTOR__
15 #define __CPHSTL_VECTOR__
16
17 #include <algorithm> // std::equal, std::lexicographical_compare,
       std::rotate
18 #include <cstddef> // std::size_t, std::ptrdiff_t
19 #include <initializer_list> // std::initializer_list
20 #include <iterator> // std::reverse_iterator
21 #include <limits> // std::numeric_limits
22 #include <memory> // std::allocator, std::allocator_traits
23 #include <stdexcept> // std::length_error, std::out_of_range
24 #include <type_traits> // std::conditional, std::true_type, std::
       false_type
25 #include <vector> // std::vector
26 #include <utility> // std::swap, std::move, std::forward
27
28 namespace cphstl {
29
30   template <typename V, typename A = std::allocator<V>,
31             typename K = std::vector<V, A>>
32   class vector {
33   public:
34
35     // types
36
37     using value_type = V;
38     using allocator_type = A;
39     using reference = typename K::reference;
40     using const_reference = typename K::const_reference;
41     using pointer = typename std::allocator_traits<A>::pointer;
42     using const_pointer = typename std::allocator_traits<A>::
           const_pointer;
43     using size_type = std::size_t;
44     using difference_type = std::ptrdiff_t;
45     using iterator = typename K::iterator;
46     using const_iterator = typename K::const_iterator;
47     using reverse_iterator = std::reverse_iterator<iterator>;
48     using const_reverse_iterator = std::reverse_iterator<
           const_iterator>;
49
50   public:
51
52     // structors
53
54     vector() noexcept;
55     explicit vector(A const&) noexcept;
56     explicit vector(size_type, A const& = A());
57     vector(size_type, V const&, A const& = A());
58
59     template <typename I>
60     vector(I, I, A const& = A());
61
```

```
62      vector(vector const&);
63      vector(vector const&, A const&);
64      vector(vector&&) noexcept;
65      vector(vector&&, A const&) noexcept(
66        std::allocator_traits<A>::
              propagate_on_container_move_assignment::value);
67  // C++17 or std::allocator_traits<A>::is_always_equal::value);
68      vector(std::initializer_list<V>, A const& = A());
69      ~vector();
70      vector& operator=(vector const&);
71      vector& operator=(vector&&) noexcept(
72        std::allocator_traits<A>::
              propagate_on_container_move_assignment::value);
73  // C++17 or std::allocator_traits<A>::is_always_equal::value);
74      vector& operator=(std::initializer_list<V>);
75
76      template <typename I>
77      void assign(I, I);
78      void assign(size_type, V const&);
79      void assign(std::initializer_list<V>);
80
81      // iterators
82
83      iterator begin() noexcept;
84      const_iterator begin() const noexcept;
85      iterator end() noexcept;
86      const_iterator end() const noexcept;
87      reverse_iterator rbegin() noexcept;
88      const_reverse_iterator rbegin() const noexcept;
89      reverse_iterator rend() noexcept;
90      const_reverse_iterator rend() const noexcept;
91
92      const_iterator cbegin() const noexcept;
93      const_iterator cend() const noexcept;
94      const_reverse_iterator crbegin() const noexcept;
95      const_reverse_iterator crend() const noexcept;
96
97      // accessors
98
99      A get_allocator() const noexcept;
100     size_type size() const noexcept;
101     size_type max_size() const noexcept;
102     size_type capacity() const noexcept;
103     bool empty() const noexcept;
104
105     const_reference operator[](size_type) const;
106     const_reference at(size_type) const;
107     const_reference front() const;
108     const_reference back() const;
109
110     // modifiers
111
```

```
112     reference operator[](size_type);
113     reference at(size_type);
114     reference front();
115     reference back();
116
117     V* data() noexcept;
118     V const* data() const noexcept;
119
120     void resize(size_type);
121     void resize(size_type, V const&);
122     void reserve(size_type);
123     void shrink_to_fit();
124
125     template <typename... Args>
126     void emplace_back(Args&&...);
127
128     void push_back(V const&);
129     void push_back(V&&);
130     void pop_back();
131
132     template <typename... Args>
133     iterator emplace(iterator, Args&&...);
134
135     iterator insert(iterator, V const&);
136     iterator insert(iterator, V&&);
137     iterator insert(iterator, size_type, V const&);
138
139     template <typename I>
140     iterator insert(iterator, I, I);
141
142     iterator insert(iterator, std::initializer_list<V>);
143     iterator erase(iterator);
144     iterator erase(iterator, iterator);
145     void swap(vector&) noexcept(
146       std::allocator_traits<A>::propagate_on_container_swap::value);
147 // C++17      or std::allocator_traits<A>::is_always_equal::value);
148     void clear() noexcept;
149
150   protected:
151
152     // helpers
153
154     template <typename S, typename I>
155     void append(S&, I, I);
156
157     template <typename S>
158     void append_n(S&, size_type, V const&);
159
160     template <typename I>
161     void assign_dispatch(I, I, std::false_type);
162
163     template <typename I>
```

```
164     void assign_dispatch(I, I, std::true_type);
165
166     void shrink_to_fit_dispatch(std::false_type);
167     void shrink_to_fit_dispatch(std::true_type);
168
169     template <typename I>
170     iterator insert_dispatch(iterator, I, I, std::false_type);
171
172     template <typename I>
173     iterator insert_dispatch(iterator, I, I, std::true_type);
174
175     iterator insert_fill(iterator, size_type, V const&);
176
177     template <typename I>
178     iterator insert_range(iterator, I, I);
179
180     using kernel_type = K;
181     kernel_type kernel;
182
183   };
184
185   template <typename V, typename A, typename K>
186   bool operator==(vector<V, A, K> const&, vector<V, A, K> const&);
187
188   template <typename V, typename A, typename K>
189   bool operator<(vector<V, A, K> const&, vector<V, A, K> const&);
190
191   template <typename V, typename A, typename K>
192   bool operator≠(vector<V, A, K> const&, vector<V, A, K> const&);
193
194   template <typename V, typename A, typename K>
195   bool operator>(vector<V, A, K> const&, vector<V, A, K> const&);
196
197   template <typename V, typename A, typename K>
198   bool operator≥(vector<V, A, K> const&, vector<V, A, K> const&);
199
200   template <typename V, typename A, typename K>
201   bool operator≤(vector<V, A, K> const&, vector<V, A, K> const&);
202
203   template <typename V, typename A, typename K>
204   void swap(vector<V, A, K>& x, vector<V, A, K>& y) noexcept(
205     noexcept(x.swap(y)));
206 }
207
208 #include "stl-vector.i++" // implementation
209
210 #endif
```

stl-vector.i++

```
1 /*
2   A vector is a container that delegates its work to the given kernel
```

```
3
4   Performance Engineering Laboratory © 2008, 2016
5 */
6
7 namespace cphstl {
8
9   // helpers
10
11  template <typename V, typename A, typename K>
12  template <typename S, typename I>
13  void vector<V, A, K>::append(S& sequence, I some, I past) {
14    I p = some;
15    I q = some;
16    try {
17      for ( ; q != past; ++q) {
18        sequence.push_back(*q);
19      }
20    }
21    catch (...) {
22      for ( ; p != q; ++p) {
23        sequence.pop_back();
24      }
25      throw;
26    }
27  }
28
29  template <typename V, typename A, typename K>
30  template <typename S>
31  void vector<V, A, K>::append_n(S& sequence, size_type k, V const&
        value) {
32    size_type i = 0;
33    size_type j = 0;
34    try {
35      for ( ; j != k; ++j) {
36        sequence.push_back(value);
37      }
38    }
39    catch (...) {
40      for ( ; i != j; ++i) {
41        sequence.pop_back();
42      }
43    }
44  }
45
46  // default constructor
47
48  template <typename V, typename A, typename K>
49  vector<V, A, K>::vector() noexcept
50    : kernel(A()) {
51  }
52
53  // explicit constructors
```

```
54
55  template <typename V, typename A, typename K>
56  vector<V, A, K>::vector(A const& a) noexcept
57    : kernel(a) {
58  }
59
60  template <typename V, typename A, typename K>
61  vector<V, A, K>::vector(size_type n, A const& a)
62    : kernel(a) {
63    append_n(kernel, n, V());
64  }
65
66  // parametrized constructors
67
68  template <typename V, typename A, typename K>
69  vector<V, A, K>::vector(size_type n, V const& v, A const& a)
70    : kernel(a) {
71    append_n(kernel, n, v);
72  }
73
74  template <typename V, typename A, typename K>
75  template <typename I>
76  vector<V, A, K>::vector(I p, I q, A const& a)
77    : kernel(a) {
78    append(kernel, p, q);
79  }
80
81  // copy constructors
82
83  template <typename V, typename A, typename K>
84  vector<V, A, K>::vector(vector const& other)
85    : kernel(other.get_allocator()) {
86    K tmp(other.get_allocator());
87    append(tmp, other.begin(), other.end());
88    kernel.swap(tmp);
89  }
90
91  template <typename V, typename A, typename K>
92  vector<V, A, K>::vector(vector const& other, A const& a)
93    : kernel(a) {
94    K tmp(a);
95    append(tmp, other.begin(), other.end());
96    kernel.swap(tmp);
97  }
98
99  // move constructors
100
101 template <typename V, typename A, typename K>
102 vector<V, A, K>::vector(vector&& other) noexcept
103   : kernel(other.get_allocator()) {
104   kernel.swap(other.kernel);
105 }
```

```
106
107   template <typename V, typename A, typename K>
108   vector<V, A, K>::vector(vector&& other, A const& a) noexcept(
109     std::allocator_traits<A>::propagate_on_container_move_assignment
          ::value)
110     : kernel(a) {
111     kernel.swap(other.kernel);
112   }
113
114   // initializer-list constructor
115
116   template <typename V, typename A, typename K>
117   vector<V, A, K>::vector(std::initializer_list<V> other, A const& a
        )
118     : kernel(a) {
119     K tmp(a);
120     append(tmp, other.begin(), other.end());
121     kernel.swap(tmp);
122   }
123
124   // destructor
125
126   template <typename V, typename A, typename K>
127   vector<V, A, K>::~vector() {
128     clear();
129   }
130
131   // copy assignment
132
133   template <typename V, typename A, typename K>
134   vector<V, A, K>& vector<V, A, K>::operator=(vector const& other)
        {
135     if (this != &other) {
136       K tmp(other.kernel);
137       kernel.swap(tmp);
138     }
139     return *this;
140   }
141
142   // move assignment
143
144   template <typename V, typename A, typename K>
145   vector<V, A, K>& vector<V, A, K>::operator=(vector&& other)
        noexcept(
146     std::allocator_traits<A>::propagate_on_container_move_assignment
          ::value) {
147 // C++17 or std::allocator_traits<A>::is_always_equal::value) {
148     if (this != &other) {
149       clear();
150       kernel.swap(other.kernel);
151     }
152     return *this;
```

```
153   }
154
155   // initializer-list assignment
156
157   template <typename V, typename A, typename K>
158   vector<V, A, K>& vector<V, A, K>::operator=(std::initializer_list<
          V> other) {
159     assign_dispatch(other.begin(), other.end(), std::false_type());
160     return *this;
161   }
162
163   // assign
164
165   template <typename V, typename A, typename K>
166   void vector<V, A, K>::assign(size_type n, V const& v) {
167     K tmp(kernel.get_allocator());
168     append_n(tmp, n, v);
169     kernel.swap(tmp);
170   }
171
172   template <typename V, typename A, typename K>
173   template <typename I>
174   void vector<V, A, K>::assign(I p, I q) {
175     assign_dispatch(p, q, typename std::conditional<std::is_integral<
          I>::value, std::true_type, std::false_type>::type());
176   }
177
178   template <typename V, typename A, typename K>
179   void vector<V, A, K>::assign(std::initializer_list<V> other) {
180     assign_dispatch(other.begin(), other.end(), std::false_type());
181   }
182
183   template <typename V, typename A, typename K>
184   template <typename Integer>
185   void vector<V, A, K>::assign_dispatch(Integer n, Integer v, std::
          true_type) {
186     K tmp(kernel.get_allocator());
187     append_n(tmp, static_cast<size_type>(n), static_cast<V>(v));
188     kernel.swap(tmp);
189   }
190
191   template <typename V, typename A, typename K>
192   template <typename I>
193   void vector<V, A, K>::assign_dispatch(I p, I q, std::false_type) {
194     K tmp(kernel.get_allocator());
195     append(tmp, p, q);
196     kernel.swap(tmp);
197   }
198
199   // begin
200
201   template <typename V, typename A, typename K>
```

```
202  typename vector<V, A, K>::iterator vector<V, A, K>::begin()
         noexcept {
203    return kernel.begin();
204  }
205
206  template <typename V, typename A, typename K>
207  typename vector<V, A, K>::const_iterator vector<V, A, K>::begin()
         const noexcept {
208    return const_iterator(kernel.begin());
209  }
210
211  // end
212
213  template <typename V, typename A, typename K>
214  typename vector<V, A, K>::iterator vector<V, A, K>::end() noexcept
         {
215    return kernel.end();
216  }
217
218  template <typename V, typename A, typename K>
219  typename vector<V, A, K>::const_iterator vector<V, A, K>::end()
         const noexcept {
220    return const_iterator(kernel.end());
221  }
222
223  // rbegin
224
225  template <typename V, typename A, typename K>
226  typename vector<V, A, K>::reverse_iterator vector<V, A, K>::rbegin
         () noexcept {
227    return reverse_iterator(end());
228  }
229
230  template <typename V, typename A, typename K>
231  typename vector<V, A, K>::const_reverse_iterator vector<V, A, K>::
         rbegin() const noexcept {
232    return const_reverse_iterator(end());
233  }
234
235  // rend
236
237  template <typename V, typename A, typename K>
238  typename vector<V, A, K>::reverse_iterator vector<V, A, K>::rend()
         noexcept {
239    return reverse_iterator(begin());
240  }
241
242  template <typename V, typename A, typename K>
243  typename vector<V, A, K>::const_reverse_iterator vector<V, A, K>::
         rend() const noexcept {
244    return const_reverse_iterator(begin());
245  }
```

```
246
247   // cbegin
248
249   template <typename V, typename A, typename K>
250   typename vector<V, A, K>::const_iterator vector<V, A, K>::cbegin()
          const noexcept {
251     return const_iterator(kernel.begin());
252   }
253
254   // cend
255
256   template <typename V, typename A, typename K>
257   typename vector<V, A, K>::const_iterator vector<V, A, K>::cend()
          const noexcept {
258     return const_iterator(kernel.end());
259   }
260
261   // crbegin
262
263   template <typename V, typename A, typename K>
264   typename vector<V, A, K>::const_reverse_iterator vector<V, A, K>::
          crbegin() const noexcept {
265     return const_reverse_iterator(end());
266   }
267
268   // crend
269
270   template <typename V, typename A, typename K>
271   typename vector<V, A, K>::const_reverse_iterator vector<V, A, K>::
          crend() const noexcept {
272     return const_reverse_iterator(begin());
273   }
274
275   // get_allocator
276
277   template <typename V, typename A, typename K>
278   A vector<V, A, K>::get_allocator() const noexcept {
279     return kernel.get_allocator();
280   }
281
282   // size
283
284   template <typename V, typename A, typename K>
285   typename vector<V, A, K>::size_type vector<V, A, K>::size() const
          noexcept {
286     return kernel.size();
287   }
288
289   // max_size
290
291   template <typename V, typename A, typename K>
292   typename vector<V, A, K>::size_type vector<V, A, K>::max_size()
```

```
    const noexcept {
293   return std::numeric_limits<difference_type>::max();
294 }
295
296 // capacity
297
298 template <typename V, typename A, typename K>
299 typename vector<V, A, K>::size_type vector<V, A, K>::capacity()
    const noexcept {
300   return kernel.capacity();
301 }
302
303 // empty
304
305 template <typename V, typename A, typename K>
306 bool vector<V, A, K>::empty() const noexcept {
307   return size() == size_type(0);
308 }
309
310 // operator []
311
312 template <typename V, typename A, typename K>
313 typename vector<V, A, K>::reference vector<V, A, K>::operator[](
    size_type i) {
314   return reference(kernel.operator[](i));
315 }
316
317 template <typename V, typename A, typename K>
318 typename vector<V, A, K>::const_reference vector<V, A, K>::operator
    [](size_type i) const {
319   return const_reference(kernel.operator[](i));
320 }
321
322 // at
323
324 template <typename V, typename A, typename K>
325 typename vector<V, A, K>::reference vector<V, A, K>::at(size_type i
    ) {
326   if (i ≥ size()) {
327     throw std::out_of_range("index out of bounds");
328   }
329   return reference(operator[](i));
330 }
331
332 template <typename V, typename A, typename K>
333 typename vector<V, A, K>::const_reference vector<V, A, K>::at(
    size_type i) const {
334   if (i ≥ size()) {
335     throw std::out_of_range("index out of bounds");
336   }
337   return const_reference(operator[](i));
338 }
```

```
339
340   // front
341
342   template <typename V, typename A, typename K>
343   typename vector<V, A, K>::reference vector<V, A, K>::front() {
344     iterator first = kernel.begin();
345     return reference(*first);
346   }
347
348   template <typename V, typename A, typename K>
349   typename vector<V, A, K>::const_reference vector<V, A, K>::front()
          const {
350     const_iterator first = kernel.begin();
351     return const_reference(*first);
352   }
353
354   // back
355
356   template <typename V, typename A, typename K>
357   typename vector<V, A, K>::reference vector<V, A, K>::back() {
358     iterator const last = --end();
359     return reference(*last);
360   }
361
362   template <typename V, typename A, typename K>
363   typename vector<V, A, K>::const_reference vector<V, A, K>::back()
          const {
364     const_iterator const last = --end();
365     return const_reference(*last);
366   }
367
368   // data
369
370   template <typename V, typename A, typename K>
371   V* vector<V, A, K>::data() noexcept {
372     return kernel.data();
373   }
374
375   template <typename V, typename A, typename K>
376   V const* vector<V, A, K>::data() const noexcept {
377     return kernel.data();
378   }
379
380   // resize
381
382   template <typename V, typename A, typename K>
383   void vector<V, A, K>::resize(size_type n) {
384     resize(n, V());
385   }
386
387   template <typename V, typename A, typename K>
388   void vector<V, A, K>::resize(size_type n, V const& v) {
```

```
389     if (size() > n) {
390       while (size() > n) {
391         pop_back();
392       }
393       return;
394     }
395     append_n(kernel, n - size(), v);
396   }
397
398   // reserve
399
400   template <typename V, typename A, typename K>
401   void vector<V, A, K>::reserve(size_type s) {
402     kernel.reserve(s);
403   }
404
405   // shrink_to_fit
406
407   template <typename T>
408   class has_shrink_to_fit {
409
410     template <typename U, void (U::*)()>
411     struct check;
412
413     template <typename U>
414     static char member(check<U, &U::shrink_to_fit>*);
415
416     template <typename U>
417     static int member(...);
418
419   public:
420
421     enum { value = sizeof(member<T>(nullptr)) == sizeof(char) };
422   };
423
424   template <typename V, typename A, typename K>
425   void vector<V, A, K>::shrink_to_fit() {
426     shrink_to_fit_dispatch(typename std::conditional<
          has_shrink_to_fit<K>::value, std::true_type, std::false_type
          >::type());
427   }
428
429   template <typename V, typename A, typename K>
430   void vector<V, A, K>::shrink_to_fit_dispatch(std::true_type) {
431     kernel.shrink_to_fit();
432   }
433
434   template <typename V, typename A, typename K>
435   void vector<V, A, K>::shrink_to_fit_dispatch(std::false_type) {
436     // do nothing
437   }
438
```

```
439  // emplace_back
440
441  template <typename V, typename A, typename K>
442  template <typename... Args>
443  void vector<V, A, K>::emplace_back(Args&&... args) {
444    kernel.emplace_back(std::forward<Args>(args)...);
445  }
446
447  // push_back
448
449  template <typename V, typename A, typename K>
450  void vector<V, A, K>::push_back(V const& v) {
451    kernel.push_back(v);
452  }
453
454  template <typename V, typename A, typename K>
455  void vector<V, A, K>::push_back(V&& v) {
456    kernel.push_back(std::move(v));
457  }
458
459  // pop_back
460
461  template <typename V, typename A, typename K>
462  void vector<V, A, K>::pop_back() {
463    kernel.pop_back();
464  }
465
466  // emplace
467
468  template <typename V, typename A, typename K>
469  template <typename... Args>
470  typename vector<V, A, K>::iterator vector<V, A, K>::emplace(
         iterator p, Args&&... args) {
471    size_type i = p - begin();
472    size_type j = size();
473    emplace_back(std::forward<Args>(args)...);
474    std::rotate(begin() + i, begin() + j, end());
475    return begin() + i;
476  }
477
478  // single-element insert
479
480  template <typename V, typename A, typename K>
481  typename vector<V, A, K>::iterator vector<V, A, K>::insert(iterator
         p, V const& v) {
482    size_type i = p - begin();
483    size_type j = size();
484    push_back(v);
485    std::rotate(begin() + i, begin() + j, end());
486    return begin() + i;
487  }
488
```

```
489  template <typename V, typename A, typename K>
490  typename vector<V, A, K>::iterator vector<V, A, K>::insert(iterator
         p, V&& v) {
491    size_type i = p - begin();
492    size_type j = size();
493    push_back(std::move(v));
494    std::rotate(begin() + i, begin() + j, end());
495    return begin() + i;
496  }
497
498  // multiple-element inserts
499
500  template <typename V, typename A, typename K>
501  typename vector<V, A, K>::iterator vector<V, A, K>::insert(iterator
         p, size_type n, V const& v) {
502    return insert_fill(p, n, v);
503  }
504
505  template <typename V, typename A, typename K>
506  template <typename I>
507  typename vector<V, A, K>::iterator vector<V, A, K>::insert(iterator
         p, I q, I r) {
508    return insert_dispatch(p, q, r, typename std::conditional<std::
           is_integral<I>::value, std::true_type, std::false_type>::type
           ());
509  }
510
511  template <typename V, typename A, typename K>
512  template <typename I>
513  typename vector<V, A, K>::iterator vector<V, A, K>::insert_dispatch
       (iterator p, I n, I v, std::true_type) {
514    return insert_fill(p, static_cast<size_type>(n), static_cast<V>(v
           ));
515  }
516
517  template <typename V, typename A, typename K>
518  template <typename I>
519  typename vector<V, A, K>::iterator vector<V, A, K>::insert_dispatch
       (iterator p, I q, I r, std::false_type) {
520    return insert_range(p, q, r);
521  }
522
523  template <typename V, typename A, typename K>
524  typename vector<V, A, K>::iterator vector<V, A, K>::insert(iterator
         p, std::initializer_list<V> ℓ) {
525    return insert_range(p, ℓ.begin(), ℓ.end());
526  }
527
528  template <typename V, typename A, typename K>
529  typename vector<V, A, K>::iterator vector<V, A, K>::insert_fill(
       iterator p, size_type n, V const& v) {
530    size_type i = p - begin();
```

```
531     size_type j = size();
532     append_n(kernel, n, v);
533     std::rotate(begin() + i, begin() + j, end());
534     return begin() + i;
535   }
536
537   template <typename V, typename A, typename K>
538   template <typename I>
539   typename vector<V, A, K>::iterator vector<V, A, K>::insert_range(
          iterator p, I q, I r) {
540     size_type i = p - begin();
541     size_type j = size();
542     append(kernel, q, r);
543     std::rotate(begin() + i, begin() + j, end());
544     return begin() + i;
545   }
546
547   // single-element erase
548
549   template <typename V, typename A, typename K>
550   typename vector<V, A, K>::iterator vector<V, A, K>::erase(iterator
          p) {
551     assert(p ≠ end());
552     size_type i = p - begin();
553     std::rotate(begin() + i, begin() + i + 1, end());
554     pop_back();
555     return begin() + i;
556   }
557
558   // multiple-element erase
559
560   template <typename V, typename A, typename K>
561   typename vector<V, A, K>::iterator vector<V, A, K>::erase(iterator
          p, iterator q) {
562     size_type i = p - begin();
563     size_type j = q - begin();
564     std::rotate(begin() + i, begin() + j, end());
565     while (i ≠ j) {
566       pop_back();
567       ++i;
568     }
569     return begin() + i;
570   }
571
572   // clear
573
574   template <typename V, typename A, typename K>
575   void vector<V, A, K>::clear() noexcept {
576     while (not empty()) {
577       pop_back();
578     }
579   }
```

```
580
581   // swap
582
583   template <typename V, typename A, typename K>
584   void vector<V, A, K>::swap(vector<V, A, K>& other) noexcept(
585     std::allocator_traits<A>::propagate_on_container_swap::value) {
586 // C++17       or std::allocator_traits<A>::is_always_equal::value) {
587     kernel.swap(other.kernel);
588   }
589
590   template <typename V, typename A, typename K>
591   void swap(vector<V, A, K>& x, vector<V, A, K>& y) noexcept(
592     noexcept(x.swap(y))) {
593     x.swap(y);
594   }
595
596   // operator ==
597
598   template <typename V, typename A, typename K>
599   bool operator==(vector<V, A, K> const& r, vector<V, A, K> const&
        s) {
600     return (r.size() == s.size() and std::equal(r.begin(), r.end(), s
          .begin()));
601   }
602
603   // operator <
604
605   template <typename V, typename A, typename K>
606   bool operator<(vector<V, A, K> const& r, vector<V, A, K> const& s
        ) {
607     return std::lexicographical_compare(r.begin(), r.end(), s.begin()
          , s.end());
608   }
609
610   // operator !=
611
612   template <typename V, typename A, typename K>
613   bool operator!=(vector<V, A, K> const& r, vector<V, A, K> const&
        s) {
614     return not (r == s);
615   }
616
617   // operator >
618
619   template <typename V, typename A, typename K>
620   bool operator>(vector<V, A, K> const& r, vector<V, A, K> const& s
        ) {
621     return (s < r);
622   }
623
624   // operator >=
625
```

```
626   template <typename V, typename A, typename K>
627   bool operator≥(vector<V, A, K> const& r, vector<V, A, K> const& s
          ) {
628     return not (r < s);
629   }
630
631   // operator ≤
632
633   template <typename V, typename A, typename K>
634   bool operator≤(vector<V, A, K> const& r, vector<V, A, K> const& s
          ) {
635     return not (s < r);
636   }
637 }
```

Helpers

rank-iterator.h++

```
1  /*
2    A rank iterator encapsulates a location of an element by storing a
3    (pointer, rank) pair; the pointer refers to a data structure,
4    called the owner, that contains the element referred to and the
5    rank is the index of that element within the data structure.
6
7    Authors: Jyrki Katajainen, Andreas Milton Maniotis, Bo Simonsen
8    © 2008, 2012, 2013
9  */
10
11 #ifndef __CPHSTL_RANK_ITERATOR__
12 #define __CPHSTL_RANK_ITERATOR__
13
14 #include <cassert>
15 #include <iterator> // std::random_access_iterator_tag
16 #include <limits> // std::numeric_limits
17 #include <type_traits> // std::conditional, std::is_const, std::
       remove_const
18
19 namespace cphstl {
20
21   template <typename R>
22   class rank_iterator {
23
24   public:
25
26     // associated types
27
28     using owner_type = R;
29     using opposite_type = typename std::conditional<std::is_const<R
           >::value, typename std::remove_const<R>::type, R const>::type
           ;
```

```
30      using size_type = typename R::size_type;
31      using difference_type = typename R::difference_type;
32      using value_type = typename R::value_type;
33      using pointer = typename std::conditional<std::is_const<R>::value
            , value_type const*, value_type*>::type;
34      using reference = typename std::conditional<std::is_const<R>::
            value, typename R::const_reference, typename R::reference>::
            type;
35      using iterator_category = std::random_access_iterator_tag;
36
37      // friends
38
39      friend R;
40
41      friend class rank_iterator<opposite_type>;
42
43    private:
44
45      // classes
46
47      template <typename X, typename Y>
48      struct is_comparable;
49
50      template <typename X>
51      struct is_comparable<X, typename std::remove_const<X>::type> {
52        class yes;
53      };
54
55      template <typename X>
56      struct is_comparable<X, typename std::add_const<X>::type> {
57        class yes;
58      };
59
60      // constants
61
62      static size_type constexpr sentinel = std::numeric_limits<
            size_type>::max();
63
64      // variables
65
66      owner_type* owner_p;
67      size_type rank;
68
69      // parameterized constructor
70
71      rank_iterator(owner_type* p, size_type offset = sentinel)
72        : owner_p(p), rank(offset) {
73      }
74
75    public:
76
77      // default constructor
```

```
78
79      rank_iterator()
80        : owner_p(nullptr), rank(sentinel) {
81      }
82
83      // copy constructors
84
85      // generated by the compiler if needed
86      // rank_iterator(const rank_iterator&) = default;
87
88      rank_iterator(rank_iterator<typename std::remove_const<R>::type>
            const& x)
89        : owner_p(x.owner_p), rank(x.rank) {
90      }
91
92      // assignments
93
94      // generated by the compiler if needed
95      // rank_iterator& operator=(rank_iterator const&) = default;
96
97      rank_iterator& operator=(rank_iterator<typename std::
            remove_const<R>::type> const& x) {
98        owner_p = x.owner_p;
99        rank = x.rank;
100       return *this;
101     }
102
103     // destructor
104
105     ~rank_iterator() {
106     };
107
108     // operator *
109
110     reference operator*() const {
111       assert(owner_p != nullptr and rank != sentinel);
112       return (*owner_p)[rank];
113     }
114
115     // operator ->
116
117     pointer operator->() const {
118       assert(owner_p != nullptr and rank != sentinel);
119       return &(*owner_p)[rank];
120     }
121
122     // operator ++; pre-increment
123
124     rank_iterator& operator++() {
125       assert(owner_p != nullptr and rank != sentinel);
126       ++rank;
127       if (rank == (*owner_p).size()) {
```

```
128         rank = sentinel;
129       }
130       return *this;
131     }
132
133     // operator ++; post-increment
134
135     rank_iterator operator++(int) {
136       rank_iterator return_value(*this);
137       operator++();
138       return return_value;
139     }
140
141     // operator --; pre-decrement
142
143     rank_iterator& operator--() {
144       assert(owner_p != nullptr and (*owner_p).size() != 0);
145       if (rank == sentinel) {
146         rank = (*owner_p).size() - 1;
147       }
148       else {
149         --rank;
150       }
151       return *this;
152     }
153
154     // operator --; post-decrement
155
156     rank_iterator operator--(int) {
157       rank_iterator return_value(*this);
158       operator--();
159       return return_value;
160     }
161
162     // operator +=
163
164     rank_iterator& operator+=(difference_type n) {
165       assert(owner_p != nullptr);
166       difference_type new_point = rank;
167       if (rank == sentinel) {
168         new_point = (*owner_p).size();
169       }
170       new_point += n;
171       if (new_point < 0) {
172         rank = sentinel;
173         return *this;
174       }
175       rank = size_type(new_point);
176       if (rank ≥ (*owner_p).size()) {
177         rank = sentinel;
178       }
179       return *this;
```

```
180     }
181
182     // operator -=
183
184     rank_iterator& operator-=(difference_type n) {
185       return operator+=(-n);
186     }
187
188     // operator +
189
190     rank_iterator operator+(difference_type n) const {
191       rank_iterator temporary = *this;
192       temporary.operator+=(n);
193       return temporary;
194     }
195
196     // operator -
197
198     rank_iterator operator-(difference_type n) const {
199       return operator+(-n);
200     }
201
202     // iterator distance
203
204     template <typename S, typename = typename is_comparable<S, R>::
            yes>
205     difference_type operator-(rank_iterator<S> const& other) const {
206       assert(owner_p == other.owner_p);
207       size_type y = rank;
208       if (rank == sentinel) {
209         y = (*owner_p).size();
210       }
211       size_type x = other.rank;
212       if (other.rank == sentinel) {
213         x = (*owner_p).size();
214       }
215       return y - x;
216     }
217
218     // operator ==
219
220     template <typename S, typename = typename is_comparable<S, R>::
            yes>
221     bool operator==(rank_iterator<S> const& other) const {
222       return rank == other.rank and owner_p == other.owner_p;
223     }
224
225     // operator !=
226
227     template <typename S>
228     bool operator!=(rank_iterator<S> const& other) const {
229       return not (*this == other);
```

```
230     }
231
232     // operator <
233
234     template <typename S>
235     bool operator<(rank_iterator<S> const& other) const {
236       return ((*this) - other) < difference_type(0);
237     }
238
239     // operator >
240
241     template <typename S>
242     bool operator>(rank_iterator<S> const& other) const {
243       return other < *this;
244     }
245
246     // operator ≤
247
248     template <typename S>
249     bool operator≤(rank_iterator<S> const& other) const {
250       return not (other < *this);
251     }
252
253     // operator ≥
254
255     template <typename S>
256     bool operator≥(rank_iterator<S> const& other) const {
257       return not (*this < other);
258     }
259   };
260
261   // operator +(int, iterator)
262
263   template <typename R>
264   rank_iterator<R>
265   operator+(typename R::difference_type n, rank_iterator<R> const& a
          ) {
266     return a + n;
267   }
268 }
269
270 #endif
```

leda-compare-functions.i++

```
1 /*
2   A predefined compare functor is provided for each type, for which
3   operator< is defined.
4
5   Authors: Jyrki Katajainen, Michael Neidhardt, Bo Simonsen © 2009,
        2016
6 */
```

```
7
8  #ifndef __LEDA_COMPARE_FUNCTIONS__
9  #define __LEDA_COMPARE_FUNCTIONS__
10
11 #include <functional> // std::binary_function
12 #include <type_traits> // std::is_convertible
13 #include <utility> // std::pair
14
15 namespace cphleda {
16
17   template <typename K, typename L = K>
18   class comparator
19     : public std::binary_function<K, L, int> {
20   public:
21
22     int operator()(K const& x, L const& y) const {
23       static_assert(std::is_convertible<L, K>::value,
24                     "Only convertible types can be compared");
25       if (x < y) {
26         return -1;
27       }
28       if (y < x) {
29           return 1;
30       }
31       return 0;
32     }
33   };
34
35   template <typename T>
36   class stl_compare_less
37     : public std::binary_function<T, T, bool> {
38   public:
39
40     stl_compare_less(int (*_f)(T const&, T const&)) {
41       (*this).f = _f;
42     }
43
44     bool operator()(T const& a, T const& b) const {
45       if ((*this).f(a, b) == -1) {
46         return true;
47       }
48       return false;
49     }
50
51   private:
52
53     int (*f)(T const&, T const&);
54   };
55
56   template <typename P, typename I>
57   class stl_compare_less_pair
58     : public std::binary_function<P, I, bool> {
```

```
59    public:
60
61      stl_compare_less_pair(int (*_f)(P const&, P const&)) {
62        (*this).f = _f;
63      }
64
65      bool operator()(std::pair<P, I> const& a, std::pair<P, I> const
            & b) const {
66        if ((*this).f(a.first, b.first) == 1) {
67          return true;
68        }
69        return false;
70      }
71
72    private:
73
74      int (*f)(P const&, P const&);
75    };
76
77    template <typename T>
78    int compare(T const& a, T const& b) {
79      if (a < b) {
80        return -1;
81      }
82      else if (a > b) {
83        return 1;
84      }
85      return 0;
86    }
87  }
88
89  #endif
```

counting-allocator.h++

```
1  /* -*- C++ -*-
2
3    An allocator that counts the number of bytes allocated and
4    delagates the actual work to std::allocator.
5
6    Define VERBOSE to get information on every allocation and
7    deallocation.
8
9    Authors: Jyrki Katajainen, Bjarke Buur Mortensen © 2001, 2012
10 */
11
12 #ifndef __CPHSTL_COUNTING_ALLOCATOR__
13 #define __CPHSTL_COUNTING_ALLOCATOR__
14
15 #include <iostream> // std streams
16 #include <memory> // std::allocator
17 #include <utility> // std::forward
```

```
18
19 // Use base class to have a single object counter
20
21 class counting_allocator_base {
22
23 public:
24
25   using count_type = unsigned long;
26
27   static count_type bytes_in_use() {
28     return current_bytes;
29   }
30
31   static count_type allocators_in_use() {
32     return allocators;
33   }
34
35   static count_type max_bytes_allocated() {
36     return max_bytes;
37   }
38
39   // reset total statistics
40
41   static void reset_counts() {
42     allocators = 0;
43     current_bytes = 0;
44     max_bytes = 0;
45   }
46
47 protected:
48
49   static count_type allocators; // number of allocators derived from
         this class
50   static count_type current_bytes; // current number of allocated
         bytes
51   static count_type max_bytes; // max number of bytes allocated
52 };
53
54 template <typename T>
55 class counting_allocator
56   : public counting_allocator_base {
57
58 private:
59
60   count_type alloc_id; // id of the allocator
61   std::allocator<T> alloc;
62   count_type elements;  // current number of elements allocated
63   count_type max_elements; // max number of elements allocated
64
65 public:
66
67   using value_type = typename std::allocator<T>::value_type;
```

```
68   using size_type = typename std::allocator<T>::size_type;
69   using difference_type = typename std::allocator<T>::difference_type
         ;
70
71   using pointer = typename std::allocator<T>::pointer;
72   using const_pointer = typename std::allocator<T>::const_pointer;
73
74   using reference = typename std::allocator<T>::reference;
75   using const_reference = typename std::allocator<T>::const_reference
         ;
76
77   pointer address(reference r) const {
78     return &r;
79   }
80
81   const_pointer address(const_reference r) const {
82     return &r;
83   }
84
85   counting_allocator() throw()
86     : alloc(), elements(0), max_elements(0) {
87     alloc_id = allocators++;
88   }
89
90   template <typename U>
91   counting_allocator(counting_allocator<U> const&) throw()
92     : alloc(), elements(0), max_elements(0) {
93     alloc_id = allocators++;
94   }
95
96   // note that, if allocation fails, we update our variables wrongly
97
98   pointer allocate(size_type n) {
99     elements += n;
100    current_bytes += n * sizeof(T);
101    if (max_elements < elements) {
102      max_elements = elements;
103    }
104    if (max_bytes < current_bytes) {
105      max_bytes = current_bytes;
106    }
107
108 #ifdef VERBOSE
109
110    std::cout << __PRETTY_FUNCTION__ << std::endl;
111    std::cout << "[" << alloc_id << "]\t" << "Allocating " << n
112        << " elements (" << n * sizeof(T) << " bytes)" << std::endl
113        << "\tTotal = " << elements << " ("
114        << elements * sizeof(T) << " bytes)" << std::endl;
115    std::cout << "[T] = " << current_bytes << " bytes" << std::endl;
116
117 #endif
```

```
118
119     return alloc.allocate(n);
120   }
121
122   void deallocate(pointer p, size_type n)  {
123     alloc.deallocate(p, n);
124     elements -= n;
125     current_bytes -= n * sizeof(T);
126
127 #ifdef VERBOSE
128
129     std::cout << __PRETTY_FUNCTION__ << std::endl;
130     std::cout << "[" << alloc_id << "]\t" << "Deallocating " << n
131         << " elements ("  << n * sizeof(T) << " bytes)" << std::endl
132         << "\tTotal = " << elements << " ("
133         << elements * sizeof(T) << " bytes)" << std::endl;
134     std::cout << "[T] = " << current_bytes << " bytes" << std::endl;
135
136 #endif
137
138   }
139
140   void construct(pointer p, const T& val) {
141     alloc.construct(p, val);
142   }
143
144   void construct(pointer p, T&& val) {
145     alloc.construct(p, std::forward<T>(val));
146   }
147
148   void destroy(pointer p) {
149     alloc.destroy(p);
150   }
151
152   size_type max_size() const throw() {
153     return alloc.max_size();
154   }
155
156   template <typename U>
157   struct rebind {
158     using other = counting_allocator<U>;
159   };
160
161   // accounting functions
162
163   count_type elements_allocated() {
164     return elements;
165   }
166
167   count_type max_elements_allocated() {
168     return max_elements;
169   }
```

```
170
171   count_type bytes_allocated() {
172     return elements_allocated() * sizeof(T);
173   }
174
175   count_type max_bytes_allocated() {
176     return max_elements_allocated() * sizeof(T);
177   }
178 };
179
180 // initialize static members
181
182 counting_allocator_base::count_type counting_allocator_base::
      allocators = 0;
183 counting_allocator_base::count_type counting_allocator_base::
      current_bytes = 0;
184 counting_allocator_base::count_type counting_allocator_base::
      max_bytes = 0;
185
186 #endif
```

Includes

swap_based.i++

```
1  #include <utility>  // std::swap
2
3  namespace swap_based {
4
5    template <typename iterator>
6    void __reverse(iterator first, iterator last) {
7      while (true) {
8        if (first == last or first == --last) {
9          return;
10       }
11       else {
12         std::swap(*first, *last);
13         ++first;
14       }
15     }
16   }
17
18   template <typename S>
19   void reverse(S& s) {
20     __reverse(s.begin(), s.end());
21   }
22 }
```

move_based.i++

```
1  #include <utility> // std::move
```

```
2
3 namespace move_based {
4
5   template <typename S, typename T>
6     void reverse_copy(S& input, T& output) {
7     auto n = input.size();
8     while (n != 0)  {
9       --n;
10      output.push_back(std::move(input[n]));
11      input.pop_back();
12    }
13  }
14
15  template <typename S>
16  void reverse(S& s) {
17    S tmp;
18    reverse_copy(s, tmp);
19    s.swap(tmp);
20  }
21 }
```

std.i++

```
1 #include <vector>
2
3 using X = std::vector<V, A>;
```

resizable_array.i++

```
1 #include "resizable_array.h++"
2
3 using X = cphstl::resizable_array<V, A>;
```

pile.i++

```
1 #include "pile.h++"
2
3 using X = cphstl::pile<V, A>;
```

sliced_array.i++

```
1 #include "sliced_array.h++"
2
3 using X = cphstl::sliced_array<V, A>;
```

space_efficient_array.i++

```
1 #include "space_efficient_array.h++"
2
3 using X = cphstl::space_efficient_array<V, A>;
```

Drivers

reverse-driver.c++

```
#if ! defined(MAXSIZE)

#define MAXSIZE (64 * 1024 * 1024)

#endif

#include <ctime> // std::clock_t, std::clock, CLOCKS_PER_SEC
#include <iostream> // std::cout, std::cerr
#include <memory> // std::allocator
#include <utility> // std::move

#include "resizable_array.h++"
#include "sliced_array.h++"
#include "space_efficient_array.h++"
#include <vector> // std::vector

#include "algorithm.i++" // ALGORITHM::reverse

#ifdef MEASURE_MOVES

long volatile moves = 0;

template <typename T>
class move_counter {
private:

  T datum;

  move_counter(move_counter const&) = delete;
  move_counter& operator=(move_counter const&) = delete;

public:

  explicit move_counter()
    : datum(0) {
    moves += 1;
  }

  template <typename number>
  explicit move_counter(number x = 0)
    : datum(x) {
    moves += 1;
  }

  move_counter(move_counter&& other) {
    datum = std::move(other.datum);
    moves += 1;
  }
```

```
49
50    move_counter& operator=(move_counter&& other) {
51      datum = std::move(other.datum);
52      moves += 1;
53      return *this;
54    }
55
56    operator T() const {
57      return datum;
58    }
59
60    template <typename U>
61    friend bool operator<(move_counter<U> const&, move_counter<U>
          const&);
62
63    template <typename U>
64    friend bool operator==(move_counter<U> const&, move_counter<U>
          const&);
65
66  };
67
68  template <typename T>
69  bool operator<(move_counter<T> const& x, move_counter<T> const& y)
        {
70    return x.datum < y.datum;
71  }
72
73  template <typename T>
74  bool operator==(move_counter<T> const& x, move_counter<T> const& y)
        {
75    return x.datum == y.datum;
76  }
77
78  #endif
79
80  void usage(int argc, char **argv) {
81    std::cerr << "Usage: " << argv[0] << " <n>\n";
82    exit(1);
83  }
84
85  int main(int argc, char** argv) {
86
87  #ifdef MEASURE_MOVES
88
89    using V = move_counter<int>;
90
91  #else
92
93    using V = int;
94
95  #endif
96
```

```
97    using A = std::allocator<V>;
98
99  #ifdef STD
100
101   using X = std::vector<V, A>;
102
103 #else
104
105   using X = cphstl::STRUCTURE<V, A>;
106
107 #endif
108
109   unsigned int n = 0;
110   if (argc == 2) {
111     n = atoi(argv[1]);
112   }
113   else {
114     usage(argc, argv);
115   }
116   if (n < 1 or n > MAXSIZE) {
117     std::cerr << "n out of bounds [1.." << MAXSIZE << "]\n";
118     usage(argc, argv);
119   };
120
121   unsigned int const repetitions = MAXSIZE / n;
122   X* many = new X[repetitions];
123   for (volatile unsigned int t = 0; t != repetitions; ++t) {
124     for (unsigned int i = 0; i != n; ++i) {
125       many[t].push_back(i);
126     }
127   }
128
129 #if defined(MEASURE_MOVES)
130
131   moves = 0;
132
133 #else
134
135   std::clock_t start = std::clock();
136
137 #endif
138
139   for (volatile unsigned int t = 0; t != repetitions; ++t) {
140     ALGORITHM::reverse(many[t]);
141   }
142
143 #if ! defined(MEASURE_MOVES)
144
145   std::clock_t stop = std::clock();
146
147 #endif
148
```

```
149   double scale = double(repetitions) * double(n);
150
151 #if defined(MEASURE_MOVES)
152
153   std::cout.precision(3);
154   std::cout << n << '\t' << double(moves) / double(scale) << std::
          endl;
155
156 #else
157
158   double ns = 1.0e9 * double(stop - start) / double(CLOCKS_PER_SEC);
159   std::cout.precision(4);
160   std::cout << n << '\t' << ns / double(scale) << std::endl;
161
162 #endif
163
164   delete[] many;
165   return 0;
166 }
```

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 <cassert> // assert macro
8 #include "counting-allocator.h++"
9 #include <cstddef> // std::size_t
10 #include <cstdlib> // random, srand, RANDMAX
11 #include <iomanip>
12 #include <iostream> // std streams
13
14 using V = int;
15 using A = counting_allocator<V>;
16
17 #include "data-structure.i++" // defines X using V and A
18
19 float run(std::size_t n) {
20   assert(RAND_MAX > n);
21   counting_allocator_base::reset_counts();
22   X container;
23   for (std::size_t i = 0; i != n; ++i) {
24     container.push_back(V(random()));
25   }
26   assert(container.size() == n);
27   return float(counting_allocator_base::bytes_in_use());
28 }
29
30 int main(int argc, char* argv[]) {
```

```
31   srand(1837362);
32   for (std::size_t n = 100000; n ≤ 10000000; n += 100000) {
33     std::cout.setf(std::ios::fixed, std::ios::floatfield);
34     std::cout.precision(3);
35     float bytes_in_use = run(n);
36     float overhead = bytes_in_use - float(n * sizeof(V));
37     float in_procents = 100.0 * overhead / float(n * sizeof(V));
38     std::cout << n << "\t" << in_procents << std::endl;
39   }
40   return 0;
41 }
```

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::sort, std::
       partial_sort
8  #include <cmath> // ilogb
9  #include <ctime> // std:.clock_t, std::clock, CLOCKS_PER_SEC
10 #include <functional> // std::less
11 #include <iostream> // std::cout, std::cerr
12 #include <iterator> // std::iterator_traits
13 #include <memory> // std::allocator
14 #include <vector> // std::vector
15
16 extern int ilogb(double) throw();
17
18 template <typename iterator>
19 bool is_permutation(iterator first, iterator beyond) {
20   using V = typename std::iterator_traits<iterator>::value_type;
21   std::vector<V> copy(first, beyond);
22   std::sort(copy.begin(), copy.end());
23   for (auto q = copy.begin(); q =/= copy.end(); ++q) {
24     V i = V(q - copy.begin());
25     if (*q =/= i) {
26       std::cerr << i << ": element missing " << *q << " instead\n";
27       std::cerr << "n: " << beyond - first << std::endl;
28       return false;
29     }
30   }
31   return true;
32 }
33
34 template <typename iterator, typename comparator>
35 bool is_sorted(iterator a, iterator o, comparator less) {
36   using Z = typename std::iterator_traits<iterator>::difference_type;
37   Z const n = o - a;
```

```
38    bool violated = false;
39    if (n < 2) {
40      return true;
41    }
42    for (Z i = n - 1; i > 0; --i) {
43      if (less(*(a + i), *(a + i - 1))) {
44        std::cerr << i << ": me " << *(a + i) << "; before " << *(a + i
              - 1) << std::endl;
45        violated = true;
46      }
47    }
48    return not violated;
49  }
50
51  using V = int;
52  using A = std::allocator<V>;
53
54  #include "data-structure.i++" // defines X using V and A
55
56  void usage(int argc, char **argv) {
57    std::cerr << "Usage: " << argv[0] << " <n>\n";
58    exit(1);
59  }
60
61  int main(int argc, char** argv) {
62    using C = std::less<V>;
63
64    unsigned int n = 0;
65    if (argc == 2) {
66      n = atoi(argv[1]);
67    }
68    else {
69      usage(argc, argv);
70    }
71    if (n < 1 or n > MAXSIZE) {
72      std::cerr << "n out of bounds [1.." << MAXSIZE << "]\n";
73      usage(argc, argv);
74    }
75
76    X b;
77    b.reserve(MAXSIZE);
78    unsigned int const repetitions = MAXSIZE / n;
79    for (unsigned int i = 0; i =/= repetitions; ++i) {
80      for (unsigned int i = 0; i =/= n; ++i) {
81        b.push_back(V(i));
82      }
83      std::random_shuffle(b.end() - n, b.end());
84    }
85
86    auto c = b.begin();
87    std::clock_t start = std::clock();
88    for (volatile unsigned int t = 0; t =/= repetitions; ++t)  {
```

```
89
90 #if defined(INTROSORT)
91
92     std::sort(c, c + n, C());
93
94 #endif
95
96 #if defined(HEAPSORT)
97
98     std::partial_sort(c, c + n, c + n, C());
99
100 #endif
101
102     c = c + n;
103   }
104   std::clock_t stop = std::clock();
105
106   auto d = b.begin();
107   for (volatile unsigned int t = 0; t != repetitions; ++t) {
108     bool ok = ::is_sorted(d, d + n, C());
109     if (! ok) {
110       return 1;
111     }
112     ok = ::is_permutation(d, d + n);
113     if (! ok) {
114       return 2;
115     }
116     d = d + n;
117   }
118
119   double scale = double(repetitions) * double(n) * double(ilogb(n));
120   double ns = 1.0e9 * double(stop - start) / double(CLOCKS_PER_SEC);
121   std::cout.precision(3);
122   std::cout << n << '\t' << ns / double(scale) << '\n';
123
124   return 0;
125 }
```

scan-driver.c++

```
1 #if ! defined(MAXSIZE)
2
3 #define MAXSIZE (128 * 1024 * 1024)
4
5 #endif
6
7 #include <cassert> // assert macro
8 #include <ctime> // std::clock_t, std::clock, CLOCKS_PER_SEC
9 #include <iostream> // std::cout, std::cerr
10 #include <memory> // std::allocator
11 #include <vector> // std::vector
12
```

```
13 using V = int;
14 using A = std::allocator<V>;
15
16 #include "data-structure.i++" // defines X using V and A
17
18 void usage(int argc, char **argv) {
19   std::cerr << "Usage: " << argv[0] << " <n>\n";
20   exit(1);
21 }
22
23 int main(int argc, char** argv) {
24   unsigned int n = 0;
25   if (argc == 2) {
26     n = atoi(argv[1]);
27   }
28   else {
29     usage(argc, argv);
30   }
31   if (n < 1 or n > MAXSIZE) {
32     std::cerr << "n out of bounds [1.." << MAXSIZE << "]\n";
33     usage(argc, argv);
34   }
35
36   std::vector<X> v;
37   unsigned int const repetitions = MAXSIZE / n;
38   v.resize(repetitions);
39   for (unsigned int t = 0; t != repetitions; ++t) {
40     // v[t].reserve(n);
41     for (unsigned int i = 0; i != n; ++i) {
42       v[t].push_back(V(i));
43     }
44   }
45
46   std::clock_t start = std::clock();
47   for (volatile unsigned int t = 0; t != repetitions; ++t)  {
48     auto c = v[t].begin();
49     auto e = v[t].end();
50     for (; c != e; ++c) {
51       *c = V(0);
52     }
53   }
54   std::clock_t stop = std::clock();
55
56   for (unsigned int t = 0; t != repetitions; ++t) {
57     for (unsigned int i = n; i != 0; ) {
58       --i;
59       assert(v[t][i] == V(0));
60       v[t].pop_back();
61     }
62   }
63
64   double ns = 1.0e9 * double(stop - start) / double(CLOCKS_PER_SEC);
```

```
65   std::cout.precision(3);
66   std::cout << n << '\t' << ns / double(repetitions * n) << '\n';
67
68   return 0;
69 }
```

jump-driver.c++

```
1  #if ! defined(MAXSIZE)
2
3  #define MAXSIZE (64 * 1024 * 1024)
4
5  #endif
6
7  #include <cassert> // assert macro
8  #include <ctime> // std::clock_t, std::clock, CLOCKS_PER_SEC
9  #include <iostream> // std::cout, std::cerr
10 #include <memory> // std::allocator
11 #include <vector> // std::vector
12
13 using V = int;
14 using A = std::allocator<V>;
15
16 #include "data-structure.i++" // defines X using V and A
17
18 void usage(int argc, char **argv) {
19   std::cerr << "Usage: " << argv[0] << " <n>\n";
20   exit(1);
21 }
22
23 int main(int argc, char** argv) {
24   unsigned int n = 0;
25   if (argc == 2) {
26     n = atoi(argv[1]);
27   }
28   else {
29     usage(argc, argv);
30   }
31   if (n < 1 or n > MAXSIZE) {
32     std::cerr << "n out of bounds [1.." << MAXSIZE << "]\n";
33     usage(argc, argv);
34   }
35
36   std::vector<X> v;
37   unsigned int const repetitions = MAXSIZE / n;
38   v.resize(repetitions);
39   for (unsigned int t = 0; t != repetitions; ++t) {
40     // v[t].reserve(n);
41     for (unsigned int i = 0; i != n; ++i) {
42       v[t].push_back(i);
43     }
44   }
```

```
45
46    unsigned int const prime = 617;
47    unsigned int const mask = n - 1;
48    std::clock_t start = std::clock();
49    for (volatile unsigned int t = 0; t =/= repetitions; ++t)  {
50      auto c = v[t].begin();
51      if (c =/= v[t].end()) {
52        *c = V(0);
53      }
54      for (unsigned int i = prime; i =/= 0; i = (i + prime) bitand mask)
            {
55        *(c + i) = V(0);
56      }
57    }
58    std::clock_t stop = std::clock();
59
60    for (unsigned int t = 0; t =/= repetitions; ++t) {
61      for (unsigned int i = n; i =/= 0; ) {
62        --i;
63        assert(v[t][i] == V(0));
64        v[t].pop_back();
65      }
66    }
67
68    double ns = 1.0e9 * double(stop - start) / double(CLOCKS_PER_SEC);
69    std::cout.precision(3);
70    std::cout << n << '\t' << ns / double(repetitions * n) << '\n';
71
72    return 0;
73  }
```

grow-driver.c++

```
1  #if ! defined(MAXSIZE)
2
3  #define MAXSIZE (128 * 1024 * 1024)
4
5  #endif
6
7  #include <ctime> // std::clock_, std::clock, CLOCKS_PER_SEC
8  #include <iostream> // std::cout, std::cerr
9  #include <memory> // std::allocator
10 #include <vector> // std::vector
11
12 using V = int;
13 using A = std::allocator<V>;
14
15 #include "data-structure.i++" // defines X using V and A
16
17 void usage(int argc, char **argv) {
18   std::cerr << "Usage: " << argv[0] << " <n>\n";
19   exit(1);
```

```
}

int main(int argc, char** argv) {
  unsigned int n = 0;
  if (argc == 2) {
    n = atoi(argv[1]);
  }
  else {
    usage(argc, argv);
  }
  if (n < 1 or n > MAXSIZE) {
    std::cerr << "n out of bounds [1.." << MAXSIZE << "]\n";
    usage(argc, argv);
  }

  std::vector<X> v;
  unsigned int const repetitions = MAXSIZE / n;
  v.resize(repetitions);

  std::clock_t start = std::clock();
  for (unsigned int t = 0; t != repetitions; ++t) {
    // v[t].reserve(n);
    for (unsigned int i = 0; i != n; ++i) {
      v[t].push_back(V(i));
    }
  }
  std::clock_t stop = std::clock();

  for (unsigned int t = 0; t != repetitions; ++t) {
    for (unsigned int i = n; i != 0; ) {
      --i;
      v[t].pop_back();
    }
  }

  double ns = 1.0e9 * double(stop - start) / double(CLOCKS_PER_SEC);
  std::cout.precision(3);
  std::cout << n << '\t' << ns / double(repetitions * n) << '\n';

  return 0;
}
```

shrink-driver.c++

```
#if ! defined(MAXSIZE)

#define MAXSIZE (128 * 1024 * 1024)

#endif

#include <ctime> // std::clock_t, std::clock, CLOCKS_PER_SEC
#include <iostream> // std::cout, std::cerr
```

```
#include <memory> // std:.allocator
#include <vector> // std::vector

using V = int;
using A = std::allocator<V>;

#include "data-structure.i++" // defines X using V and A

void usage(int argc, char **argv) {
  std::cerr << "Usage: " << argv[0] << " <n>\n";
  exit(1);
}

int main(int argc, char** argv) {
  unsigned int n = 0;
  if (argc == 2) {
    n = atoi(argv[1]);
  }
  else {
    usage(argc, argv);
  }
  if (n < 1 or n > MAXSIZE) {
    std::cerr << "n out of bounds [1.." << MAXSIZE << "]\n";
    usage(argc, argv);
  }

  std::vector<X> v;
  unsigned int const repetitions = MAXSIZE / n;
  v.resize(repetitions);
  for (unsigned int t = 0; t != repetitions; ++t) {
    // v[t].reserve(n);
    for (unsigned int i = 0; i != n; ++i) {
      v[t].push_back(V(i));
    }
  }

  std::clock_t start = std::clock();
  for (unsigned int t = 0; t != repetitions; ++t) {
    for (unsigned int i = n; i != 0; ) {
      --i;
      v[t].pop_back();
    }
  }
  clock_t stop = std::clock();

  double ns = 1.0e9 * double(stop - start) / double(CLOCKS_PER_SEC);
  std::cout.precision(3);
  std::cout << n << '\t' << ns / double(repetitions * n) << '\n';

  return 0;
}
```

many-driver.c++

```
1  /*
2    Advice: A good guess is a must!
3  */
4
5  #include <iostream> // standard streams
6  #include <memory> // std::allocator
7  #include <vector> // std::vector
8
9  using V = int;
10 using A = std::allocator<V>;
11
12 #include "data-structure.i++" // defines X using V and A
13
14 void usage(int argc, char **argv) {
15   std::cerr << "Usage: " << argv[0] << " [<guess>]\n";
16   exit(1);
17 }
18
19 int main(int argc, char** argv) {
20   unsigned int mega = 1024 * 1024;
21   std::vector<X> v;
22   unsigned int t = 0;
23   if (argc == 1) {
24   }
25   else if (argc == 2) {
26     t = atoi(argv[1]);
27   }
28   else {
29     usage(argc, argv);
30   }
31
32   while (true) {
33     unsigned int n = 0;
34     try {
35       for ( ; n != t; ++n) {
36         v.push_back(std::move(X()));
37         v[n].push_back(V(n));
38       }
39     }
40     catch (...) {
41       for (unsigned int i = 0; i != n; ++i) {
42         v[i].clear();
43       }
44       v.clear();
45       t = t - mega;
46       continue;
47     }
48     break;
49   }
```

```
50
51   while (true) {
52     try {
53       std::cout << t << std::endl;
54       v.push_back(std::move(X()));
55       for (unsigned int n = 0; n != mega; ++n) {
56         v[t].push_back(V(n));
57       }
58       for (unsigned int n = 0; n != mega - 1; ++n) { // leave one!
59         v[t].pop_back();
60       }
61       ++t;
62     }
63     catch (...) {
64       std::cerr << t << std::endl;
65       break;
66     }
67   }
68   return 0;
69 }
```

gap-driver.c++

```
1  #if ! defined(MAXSIZE)
2
3  #define MAXSIZE (128 * 1024 * 1024)
4
5  #endif
6
7  #include <ctime> // std::clock_t, std::clock, CLOCKS_PER_SEC
8  #include <iostream> // std::cout, std::cerr
9  #include <memory> // std::allocator
10 #include <vector> // std::vector
11
12 using V = int;
13 using A = std::allocator<V>;
14
15 #include "data-structure.i++" // defines X using V and A
16
17 void usage(int argc, char **argv) {
18   std::cerr << "Usage: " << argv[0] << " <n>\n";
19   exit(1);
20 }
21
22 int main(int argc, char** argv) {
23   if (argc != 2) {
24     usage(argc, argv);
25   }
26   unsigned int n = atoi(argv[1]);
27   if (n < 1 or n > MAXSIZE) {
28     std::cerr << "n out of bounds [1.." << MAXSIZE << "]\n";
29     exit(2);
```

```
30  }
31
32  std::vector<X> v;
33  unsigned int const repetitions = MAXSIZE / n;
34  v.resize(repetitions);
35  std::vector<unsigned int> gap;
36
37  for (unsigned int t = 0; t != repetitions ; ++t) {
38    V* previous = nullptr;
39    for (unsigned int i = 0; i != n; ++i) {
40      v[t].push_back(V(i));
41      V* current = &v[t].back();
42      if ((current - previous) > 4 or (previous - current) > 4) {
43        gap.push_back(i);
44      }
45      previous = current;
46    }
47  }
48
49  unsigned int gaps_per_case = gap.size() / repetitions;
50  std::cout << "n, gaps: " << n << ", " << gaps_per_case << '\n';
51  unsigned int const rounds = 20 * n / gaps_per_case;
52
53  auto p = gap.begin();
54  std::clock_t start = std::clock();
55  for (unsigned int t = 0; t != repetitions ; ++t) {
56    for (unsigned int i = 0; i != n; ++i) {
57      v[t].push_back(V(i));
58      if (i == *p) {
59        for (volatile unsigned int j = 0; j != rounds; ++j) {
60          v[t].pop_back();
61          v[t].push_back(V(i));
62        }
63        ++p;
64      }
65    }
66    while (v[t].size() > 0) {
67      v[t].pop_back();
68    }
69  }
70  std::clock_t stop = std::clock();
71
72  double scale =  double(21 * n * repetitions);
73  double ns = 1.0e9 * double(stop - start) / double(CLOCKS_PER_SEC);
74  std::cout.precision(3);
75  std::cout << n << '\t' << ns / scale << '\n';
76
77  return 0;
78 }
```

Makefile

makefile

```
CXX=g++
CXXFLAGS=-O3 -Wall -DNDEBUG -std=c++11 -msse4.2 -mabm

includes:= $(wildcard *.i++)
arrays:= $(basename $(includes))
swap:= $(addsuffix .swap-reverse, $(arrays))
move:= $(addsuffix .move-reverse, $(arrays))
space:= $(addsuffix .space, $(arrays))
introsort:= $(addsuffix .introsort, $(arrays))
heapsort:= $(addsuffix .heapsort, $(arrays))
scan:= $(addsuffix .scan, $(arrays))
jump:= $(addsuffix .jump, $(arrays))
grow:= $(addsuffix .grow, $(arrays))
shrink:= $(addsuffix .shrink, $(arrays))
many:= $(addsuffix .many, $(arrays))
gap:= $(addsuffix .gap, $(arrays))

N = 1024 32768 1048576 33554432

# Variants for reversals: -DMEASURE_MOVES -DSTD

$(swap): %.swap-reverse : %.i++
  @cp swap_based.i++ algorithm.i++
  $(CXX) $(CXXFLAGS) -DALGORITHM=swap_based -DSTRUCTURE=$* reverse-
      driver.c++
  @for n in $(N) ; do \
    ./a.out $$n ; \
  done; \
  rm -f ./a.out algorithm.i++

$(move): %.move-reverse : %.i++
  @cp move_based.i++ algorithm.i++
  $(CXX) $(CXXFLAGS) -DALGORITHM=move_based -DSTRUCTURE=$* reverse-
      driver.c++
  @for n in $(N) ; do \
    ./a.out $$n ; \
  done; \
  rm -f ./a.out algorithm.i++

$(space): %.space : %.i++
  @echo "#" $* "space-driver.c++"
  @cp $*.i++ data-structure.i++
  @$(CXX) $(CXXFLAGS) space-driver.c++
  @./a.out;
  @rm -f ./a.out data-structure.i++

$(introsort): %.introsort : %.i++
  @cp $*.i++ data-structure.i++
```

```
47   $(CXX) $(CXXFLAGS) -DINTROSORT sort-driver.c++
48   @for n in $(N) ; do \
49     ./a.out $$n ; \
50   done; \
51   rm -f ./a.out data-structure.i++
52
53 $(heapsort): %.heapsort : %.i++
54   @cp $*.i++ data-structure.i++
55   $(CXX) $(CXXFLAGS) -DHEAPSORT sort-driver.c++
56   @for n in $(N) ; do \
57     ./a.out $$n ; \
58   done; \
59   rm -f ./a.out data-structure.i++
60
61 $(scan): %.scan : %.i++
62   @cp $*.i++ data-structure.i++
63   $(CXX) $(CXXFLAGS) scan-driver.c++
64   @for n in $(N) ; do \
65     ./a.out $$n ; \
66   done; \
67   rm -f ./a.out data-structure.i++
68
69 $(jump): %.jump : %.i++
70   @cp $*.i++ data-structure.i++
71   $(CXX) $(CXXFLAGS) jump-driver.c++
72   @for n in $(N) ; do \
73     ./a.out $$n ; \
74   done; \
75   rm -f ./a.out data-structure.i++
76
77 $(grow): %.grow : %.i++
78   @cp $*.i++ data-structure.i++
79   $(CXX) $(CXXFLAGS) grow-driver.c++
80   @for n in $(N) ; do \
81     ./a.out $$n ; \
82   done; \
83   rm -f ./a.out data-structure.i++
84
85 $(shrink): %.shrink : %.i++
86   @cp $*.i++ data-structure.i++
87   $(CXX) $(CXXFLAGS) shrink-driver.c++
88   @for n in $(N) ; do \
89     ./a.out $$n ; \
90   done; \
91   rm -f ./a.out data-structure.i++
92
93 $(many): %.many : %.i++
94   @cp $*.i++ data-structure.i++
95   $(CXX) $(CXXFLAGS) many-driver.c++
96   ./a.out
97   rm -f ./a.out data-structure.i++
98
```

```
99 $(gap): %.gap : %.i++
100   @cp $*.i++ data-structure.i++
101   $(CXX) $(CXXFLAGS) gap-driver.c++
102   @for n in $(N) ; do \
103     ./a.out $$n ; \
104   done; \
105   rm -f ./a.out data-structure.i++
106
107 clean:
108   - rm  -f temp core a.out algorithm.i++ data-structure.i++ *.o 2>/
        dev/null
109
110 veryclean: clean
111   - rm -f *~ */*~ 2>/dev/null
112
113 find:
114   find . -type f -print -exec grep $(word) {} \; | less
```