Dynamic-array kernels
Katajainen, Jyrki

Publication date:
2016

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

Citation for published version (APA):
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
Copyright notice

Copyright © 2000–2016 by The authors and Performance Engineering Laboratory (University of Copenhagen)

The programs included in the CPH STL are placed in the public domain. The files may be freely copied and distributed, provided that no changes whatsoever are made. Changes are permissible only if the modified files are given new names, different from the names of existing files in the CPH STL, and only if the modified files are clearly identified as not being part of the library. Usage of the source code in derived works is otherwise unrestricted.

The authors have tried to produce correct and useful programs, but no warranty of any kind should be assumed.

Release date

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
Design overview

<table>
<thead>
<tr>
<th>Design overview</th>
<th>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</th>
</tr>
</thead>
</table>
| `cphstd::vector` | + `V`: value type  
|                  | + `A`: allocator type  
|                  | + `N`: size type  
|                  | + default constructor  
|                  | + destructor  
|                  | + `get_allocator()` const → `A`  
|                  | + `begin()` → `I`  
|                  | + `end()` → `I`  
|                  | + `size()` const → `N`  
|                  | + `reserve()` → `void`  
|                  | + `operator[](I)` → `V` const&  
|                  | + `operator[](I)` → `V`  
|                  | + `insert(I, V const&)` → `I`  
|                  | + `erase(I)` → `I`  
|                  | + // many other members |

| `cphstd::array` | + `V`: value type  
|                | + `I`: iterator type  
|                | + `item`: `I`  
|                | + `N`: size type  
|                | + default constructor  
|                | + destructor  
|                | + `operator[]()` const → `V`  
|                | + `operator[]` const → `V`  
|                | + `index_to_address()` const → `V*`  
|                | + default constructor  
|                | + `operator[]`  
|                | + `operator[]`  
|                | + `operator[]`  
|                | + // some other members |

| `rank iterator` | + `T`: owner type  
|                 | – `T`: any compatible value type  
|                 | – `Argvs`: any argument-pack type  
|                 | + default constructor  
|                 | + destructor  
|                 | + `allocate(N, void* = nullptr)` → `V*`  
|                 | + `deallocate(V*)` → `void`  
|                 | + `construct(Argvs)` → `V`  
|                 | + `destroy(V*)` → `void`  
|                 | // a few other members |

[Diagram showing class relationships and member functions of `cphstd::vector` and `cphstd::array` with `cphstd::vector` as a subclass of `cphstd::array`, illustrating the design of the dynamic-array part of the CPH STL.]

+ copy/move constructor  
+ copy/move assignment
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

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>contiguous_array.h++</td>
<td>Fixed-capacity contiguous array</td>
<td>7</td>
</tr>
<tr>
<td>resizable_array.h++</td>
<td>Resizable array</td>
<td>10</td>
</tr>
<tr>
<td>pile.h++</td>
<td>Pile of exponentially-increasing fixed-capacity slices</td>
<td>18</td>
</tr>
<tr>
<td>sliced_array.h++</td>
<td>Sliced array with fixed-capacity slices</td>
<td>24</td>
</tr>
<tr>
<td>fixed_capacity_hat.h++</td>
<td>Hashed array tree, the capacity of which is fixed</td>
<td>30</td>
</tr>
<tr>
<td>space_efficient_array.h++</td>
<td>Pile of exponentially-increasing fixed-capacity hashed array trees</td>
<td>35</td>
</tr>
<tr>
<td>leda-array.h++</td>
<td>Interface of cphleda::array</td>
<td>41</td>
</tr>
<tr>
<td>leda-array.i++</td>
<td>Implementation of cphleda::array</td>
<td>44</td>
</tr>
<tr>
<td>stl-vector.h++</td>
<td>Interface of cphstl::vector</td>
<td>52</td>
</tr>
<tr>
<td>stl-vector.i++</td>
<td>Implementation of cphstl::vector</td>
<td>56</td>
</tr>
<tr>
<td>rank_iterator.h++</td>
<td>Rank iterator</td>
<td>70</td>
</tr>
<tr>
<td>leda-compare-functions.i++</td>
<td>Comparators used by cphleda::array</td>
<td>75</td>
</tr>
<tr>
<td>counting-allocator.h++</td>
<td>Counting allocator</td>
<td>77</td>
</tr>
<tr>
<td>swap_based.i++</td>
<td>Swap-based reversal</td>
<td>81</td>
</tr>
<tr>
<td>move_based.i++</td>
<td>Move-based reversal</td>
<td>81</td>
</tr>
<tr>
<td>std.i++</td>
<td>Use of std::vector</td>
<td>82</td>
</tr>
<tr>
<td>resizable_array.i++</td>
<td>Use of cphstl::resizable_array</td>
<td>82</td>
</tr>
<tr>
<td>pile.i++</td>
<td>Use of cphstl::pile</td>
<td>82</td>
</tr>
<tr>
<td>sliced_array.i++</td>
<td>Use of cphstl::sliced_array</td>
<td>82</td>
</tr>
<tr>
<td>space_efficient_array.i++</td>
<td>Use of cphstl::space_efficient_array</td>
<td>82</td>
</tr>
<tr>
<td>reverse-driver.c++</td>
<td>Driver to run the reversal tests</td>
<td>83</td>
</tr>
<tr>
<td>space-driver.c++</td>
<td>Driver to run the space tests</td>
<td>86</td>
</tr>
<tr>
<td>sort-driver.c++</td>
<td>Driver to run the sort tests</td>
<td>87</td>
</tr>
<tr>
<td>scan-driver.c++</td>
<td>Driver to run the sequential-access iterator tests</td>
<td>89</td>
</tr>
<tr>
<td>jump-driver.c++</td>
<td>Driver to run the random-access iterator tests</td>
<td>91</td>
</tr>
<tr>
<td>grow-driver.c++</td>
<td>Driver to run the growth tests</td>
<td>92</td>
</tr>
<tr>
<td>shrink-driver.c++</td>
<td>Driver to run the shrinkage tests</td>
<td>93</td>
</tr>
<tr>
<td>many-driver.c++</td>
<td>Driver to run the break-down tests</td>
<td>95</td>
</tr>
<tr>
<td>gap-driver.c++</td>
<td>Driver to run the gap-crossing tests</td>
<td>96</td>
</tr>
<tr>
<td>makefile</td>
<td>This makefile can be used to redo the experiments described in the paper</td>
<td>98</td>
</tr>
</tbody>
</table>
References


Kernels

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

#include <cassert> // assert macro
#include <cstdlib> // std::size_t and std::ptrdiff_t
#include <memory> // std::allocator, std::uninitialized_copy/swap
#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) {

```cpp
        tmp = allocator.allocate(n);
        std::uninitialized_copy(other.X, other.X + n, tmp);
    }
    X = tmp;
    X_size = other.size();
    X_capacity = n;
}

contiguous_array(contiguous_array&& other)
    : contiguous_array(otherallocator) {
    swap(other);
}

~contiguous_array() {
    clear();
}

contiguous_array& operator=(contiguous_array const& other) {
    if (this != &other) {
        contiguous_array tmp(other);
        swap(tmp);
    }
    return *this;
}

contiguous_array& operator=(contiguous_array&& other) {
    if (this != &other) {
        contiguous_array tmp(std::move(other));
        swap(tmp);
    }
    return *this;
}

void swap(contiguous_array& other) {
    std::swap(allocator, other.allocator);
    std::swap(X, other.X);
    std::swap(X_size, other.X_size);
    std::swap(X_capacity, other.X_capacity);
}

const_iterator begin() const {
    return X;
}

iterator begin() {
    return X;
}

const_iterator end() const {
    return X + X_size;
}
```
iterator end() {
    return X + X_size;
}

V* data() noexcept {
    return X;
}

V const* data() const noexcept {
    return X;
}

allocator_type get_allocator() const {
    return allocator;
}

size_type size() const {
    return X_size;
}

void resize(size_type n) {
    assert(n <= capacity());
    if (n <= X_size) {
        for (size_type i = n; i < X_size; ++i) {
            allocator.destroy(X + i);
        }
    } else {
        std::uninitialized_fill(X + X_size, X + n, V());
    }
    X_size = n;
}

size_type capacity() const {
    return X_capacity;
}

void reserve(size_type n) {
    assert(X == nullptr); // capacity can only be set once
    X = allocator.allocate(n);
    X_size = 0;
    X_capacity = n;
}

const_reference operator[](size_type i) const {
    return X[i];
}

reference operator[](size_type i) {
    return X[i];
}
template <typename... Args>
void emplace_back(Args&&... args) {
  assert(X_size < X_capacity);
  new (X + X_size) V(std::forward<Args>(args));
  ++X_size;
}

void push_back(V const& x) {
  assert(X_size < X_capacity);
  allocator.construct(X + X_size, x);
  ++X_size;
}

void push_back(V&& x) {
  assert(X_size < X_capacity);
  allocator.construct(X + X_size, std::move(x));
  ++X_size;
}

void pop_back() {
  assert(X_size != 0);
  allocator.destroy(X + X_size);
  --X_size;
}

void clear() {
  for (size_type i = 0; i != X_size; ++i) {
    allocator.destroy(X + i);
  }
  X_size = 0;
  if (X != nullptr) {
    allocator.deallocate(X, X_capacity);
  }
  X = nullptr;
  X_capacity = 0;
};

resizable_array.h++

This is an experimental implementation of a resizable array based on doubling, halving, and incremental copying.


*/

#ifndef __CPHSTL_RESIZABLE_ARRAY__
#define __CPHSTL_RESIZABLE_ARRAY__

#include <algorithm> // std::max
```cpp
#include <cassert> // assert macro
#include <cstdlib> // 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;
```
try {
    for ( ; q != past; ++q) {
        push_back(*q);
    }
}

catch (...) {
    for ( ; p != q; ++p) {
        pop_back();
    }
    throw;
}

void append_n(size_type k, V const& value) {
    size_type i = 0;
    size_type j = 0;
    try {
        for ( ; j != k; ++j) {
            push_back(value);
        }
    }
    catch (...) {
        for ( ; i != j; ++i) {
            pop_back();
        }
    }
}

public:

explicit resizable_array(A const& a = A())
    : allocator(a), X(nullptr), Y(nullptr), X_size(0), Y_size(0),
      X_capacity(0), Y_capacity(0) {
    X = allocator.allocate(1);
    X_capacity = 1;
}

resizable_array(std::initializer_list<V> s)
    : resizable_array(A()) {
    append(s.begin(), s.end());
}

resizable_array(size_type n, A const& a)
    : resizable_array(a) {
    append_n(n, V());
}

resizable_array(resizable_array const& other)
    : resizable_array(other.get_allocator()) {
    V* tmp = nullptr;
    size_type n = other.capacity();
    tmp = allocator.allocate(n);
std::uninitialized_copy(other.begin(), other.begin() + n, tmp);
X = tmp;
X_size = other.size();
X_capacity = n;
}

resizable_array(resizable_array&& other)
: resizable_array(other.get_allocator()) {
    swap(other);
}

~resizable_array() {
    clear();
    if (X != nullptr) {
        allocator.deallocate(X, X_capacity);
    }
}

resizable_array& operator=(resizable_array const& other) {
    if (this != &other) {
        resizable_array tmp(other);
        swap(tmp);
    }
    return *this;
}

resizable_array& operator=(resizable_array&& other) {
    if (this != &other) {
        resizable_array tmp(std::move(other));
        swap(tmp);
    }
    return *this;
}

void swap(resizable_array& other) {
    std::swap(allocator, other.allocator);
    std::swap(X, other.X);
    std::swap(X_size, other.X_size);
    std::swap(X_capacity, other.X_capacity);
    std::swap(Y, other.Y);
    std::swap(Y_size, other.Y_size);
    std::swap(Y_capacity, other.Y_capacity);
}

const_iterator cbegin() const {
    return (size() == 0) ? const_iterator(this, 0) : const_iterator(this);
}

const_iterator begin() const {
    return cbegin();
}
iterator begin() {  
    return (size() ≠ 0) ? iterator(this, 0) : iterator(this);
}

const_iterator cend() const {
    return const_iterator(this);
}

const_iterator end() const {
    return cend();
}

iterator end() {
    return iterator(this);
}

allocator_type get_allocator() const {
    return allocator;
}

size_type size() const {
    return std::max(X_size, Y_size);
}

void resize(size_type n) {
    if (size() > n) {
        while (size() > n) {
            pop_back();
        }
        return;
    }
    append_n(n - size(), V());
}

size_type capacity() const {
    if (Y_size ≠ 0) {
        return Y_capacity;
    }
    return X_capacity;
}

void reserve(size_type n) {
    if (n ≤ capacity()) {
        return;
    }
    V* tmp = allocator.allocate(n);
    std::uninitialized_copy(std::make_move_iterator(X), std::make_move_iterator(X + X_size), tmp);
    if (Y ≠ nullptr) {
        std::uninitialized_copy(std::make_move_iterator(Y + X_size),
                                std::make_move_iterator(Y + Y_size), tmp + X_size);
    }
allocator.deallocate(X, X_capacity);
if (Y \neq nullptr) {
  allocator.deallocate(Y, Y_capacity);
}
X_size = size();
X_capacity = n;
X = tmp;
Y = nullptr;
Y_size = 0;
Y_capacity = 0;
}

const_reference operator[](size_type i) const {
  return *index_to_address(i);
}

reference operator[](size_type i) {
  return *index_to_address(i);
}

const_reference front() const {
  return *index_to_address(0);
}

reference front() {
  return *index_to_address(0);
}

const_reference back() const {
  return *index_to_address(size() - 1);
}

reference back() {
  return *index_to_address(size() - 1);
}

template <typename... Args>
void emplace_back(Args&&... args) {
  if (Y_size == 0 && X_size < X_capacity) {
    new (X + X_size) V(std::forward<Args>(args)...);
    ++X_size;
    return;
  }
  if (Y_size == 0 && X_size == X_capacity) {
    Y_capacity = 2 * X_capacity;
    Y = allocator.allocate(Y_capacity);
    Y_size = X_size;
  }
  --X_size;
  allocator.construct(Y + X_size, std::move(*(X + X_size)));
  allocator.destroy(X + X_size);
new (Y + Y_size) V(std::forward<Args>(args)...) + Y_size;
if (X_size == 0) {
    allocator.deallocate(X, X_capacity);
    X = Y;
    X_size = Y_size;
    X_capacity = Y_capacity;
    Y = nullptr;
    Y_size = 0;
    Y_capacity = 0;
}
}

void push_back(V const& x) {
    if (Y_size == 0 and X_size < X_capacity) {
        allocator.construct(X + X_size, x);
        ++X_size;
        return;
    }
    if (Y_size == 0 and X_size == X_capacity) {
        Y_capacity = 2 * X_capacity;
        Y = allocator.allocate(Y_capacity);
        Y_size = X_size;
    }
    --X_size;
    allocator.construct(Y + X_size, std::move(*(X + X_size)));
    allocator.destroy(X + X_size);
    allocator.construct(Y + Y_size, x);
    ++Y_size;
    if (X_size == 0) {
        allocator.deallocate(X, X_capacity);
        X = Y;
        X_size = Y_size;
        X_capacity = Y_capacity;
        Y = nullptr;
        Y_size = 0;
        Y_capacity = 0;
    }
}

void push_back(V&& x) {
    if (Y_size == 0 and X_size < X_capacity) {
        allocator.construct(X + X_size, std::move(x));
        ++X_size;
        return;
    }
    if (Y_size == 0 and X_size == X_capacity) {
        Y_capacity = 2 * X_capacity;
        Y = allocator.allocate(Y_capacity);
        Y_size = X_size;
    }
    --X_size;
allocator.construct(Y + X_size, std::move(*X));
 allocator.destroy(X + X_size);
 allocator.construct(Y + Y_size, std::move(x));
 ++Y_size;
 if (X_size == 0) {
   allocator.deallocate(X, X_capacity);
   X = Y;
   X_size = Y_size;
   X_capacity = Y_capacity;
   Y = nullptr;
   Y_size = 0;
   Y_capacity = 0;
 }
 void pop_back() {
   assert(size() > 0);
   if (Y_size == 0 and 4 * X_size > X_capacity) {
     --X_size;
     allocator.destroy(X + X_size);
     return;
   } if (Y_size == 0 and 4 * X_size <= X_capacity) {
     assert(Y_capacity == 0);
     Y_capacity = 2 * X_size;
     Y = allocator.allocate(Y_capacity);
     Y_size = X_size;
   } --X_size;
   allocator.construct(Y + X_size, std::move(*X));
   allocator.destroy(X + X_size);
   if (X_size != 0) {
     --X_size;
     allocator.construct(Y + X_size, std::move(*X));
     allocator.destroy(X + X_size);
   }
   --Y_size;
   allocator.destroy(Y + Y_size);
   if (X_size == 0) {
     allocator.deallocate(X, X_capacity);
     X = Y;
     X_size = Y_size;
     X_capacity = Y_capacity;
     Y = nullptr;
     Y_size = 0;
     Y_capacity = 0;
   }
 void clear() {
   for (size_type i = 0; i != X_size; ++i) {
     allocator.destroy(X + i);
for (size_type j = 0; j \neq Y\_size; ++j) {
    allocator.destroy(Y + j);
}
if (X \neq nullptr) {
    allocator.deallocate(X, X\_capacity);
}
if (Y \neq nullptr) {
    allocator.deallocate(Y, Y\_capacity);
}
X = allocator.allocate(1);
Y = nullptr;
X\_size = 0;
Y\_size = 0;
X\_capacity = 1;
Y\_capacity = 0;
}
};
}
#endif

pile.h++

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

*/

#ifndef __CPHSTL_PILE__
#define __CPHSTL_PILE__

#include <algorithm>
#include <cassert>
#include <cstddef>
#include <initializer_list>
#include <memory>
#include "rank-iterator.h++"
#include "resizable_array.h++"
#include <utility>

#ifndef __POPULATION_COUNT__
#define __POPULATION_COUNT__

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

unsigned long population_count(unsigned long j) {


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

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

```cpp
namespace cphstl {

    template <typename V, typename A = std::allocator<V>>
    class pile {
    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 = pile<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:
        using other_allocator_type = typename A::template rebind<V*>::other;
        using directory_type = cphstl::resizable_array<V*,
            other_allocator_type>;
    }
```
allocator_type allocator;
directory_type directory;
size_type n;

V* index_to_address(size_type rank) const {
    if (rank < 2) {
        return directory[0] + rank;
    }
    std::size_t h = whole_number_logarithm(rank);
    return directory[h] + rank - (1 << h);
}

template <typename I>
void append(I first, I past) {
    I p = first;
    I q = first;
    try {
        for (; q != past; ++q) {
            push_back(*q);
        }
    }
    catch (...) {
        for (; p != q; ++p) {
            pop_back();
        }
        throw;
    }
}

void append_n(size_type k, V const& value) {
    size_type i = 0;
    size_type j = 0;
    try {
        for (; j != k; ++j) {
            push_back(value);
        }
    }
    catch (...) {
        for (; i != j; ++i) {
            pop_back();
        }
    }
}

public:

explicit pile(A const& a = A())
    : allocator(a), directory(a), n(0) {
    V* p = allocator.allocate(2);
    directory.push_back(p);
}
pile(std::initializer_list<V> s)
 : pile(A()) {
   append(s.begin(), s.end());
}

pile(size_type n, A const& a)
 : pile(a) {
   append_n(n, V());
}

pile(pile const& other)
 : allocator(other.get_allocator()), directory(other.
   get_allocator()),
   n(other.size()) {
   size_type length = 2;
   size_type i = 0;
   do {
      V∗ s = allocator.allocate(length);
      size_type j = std::min(i + length, n);
      std::uninitialized_copy(other.begin() + i, other.begin() + j, s);
      directory.push_back(s);
      length = 1 << directory.size();
      i = j;
   } while (i != n);
}

template <typename S>
pile(S const& other)
 : allocator(other.get_allocator()), directory(other.
   get_allocator()),
   n(other.size()) {
   size_type length = 2;
   size_type i = 0;
   do {
      V∗ s = allocator.allocate(length);
      size_type j = std::min(i + length, n);
      std::uninitialized_copy(other.begin() + i, other.begin() + j, s);
      directory.push_back(s);
      length = 1 << directory.size();
      i = j;
   } while (i != n);
}

pile(pile&& other)
 : allocator(other.get_allocator()), directory(other.
   get_allocator()), n(0) {
   swap(other);
}

~pile() {

clear();
directory.clear();
}
pile& operator=(pile const& other) {
    if (this != &other) {
        pile tmp(other);
        swap(tmp);
    }
    return *this;
}
pile& operator=(pile&& other) {
    if (this != &other) {
        pile tmp(std::move(other));
        swap(tmp);
    }
    return *this;
}

const_iterator cbegin() const {
    return (size() != 0) ? const_iterator(this, 0) : const_iterator(this);
}
const_iterator begin() const {
    return cbegin();
}

iterator begin() {
    return (size() != 0) ? iterator(this, 0) : iterator(this);
}
const_iterator cend() const {
    return const_iterator(this);
}
const_iterator end() const {
    return cend();
}

iterator end() {
    return iterator(this);
}

allocator_type get_allocator() const {
    return allocator;
}

size_type size() const {
    return n;
}
size_type capacity() const {
    return 1 << directory.size();
}

void resize(size_type new_size) {
    if (n > new_size) {
        while (n > new_size) {
            pop_back();
        }
        return;
    }
    append_n(new_size - n, V());
}

void reserve(size_type) {
    // do nothing
}

const_reference operator[](size_type i) const {
    return *index_to_address(i);
}

reference operator[](size_type i) {
    return *index_to_address(i);
}

const_reference front() const {
    return *index_to_address(0);
}

reference front() {
    return *index_to_address(0);
}

const_reference back() const {
    return *index_to_address(size() - 1);
}

reference back() {
    return *index_to_address(size() - 1);
}

template <typename... Args>
void emplace_back(Args&&... args) {
    if (n > 1 and population_count(n) == 1) {
        size_type h = directory.size();
        V* p = allocator.allocate(1 << h);
        directory.push_back(p);
    }
    new (index_to_address(n)) V(std::forward<Args>(args)...);
    n += 1;
void push_back(V const& x) {
    if (n > 1 && population_count(n) == 1) {
        size_type h = directory.size();
        V* p = allocator.allocate(1 << h);
        directory.push_back(p);
    }
    allocator.construct(index_to_address(n), x);
    n += 1;
}

void push_back(V&& x) {
    if (n > 1 && population_count(n) == 1) {
        size_type h = directory.size();
        V* p = allocator.allocate(1 << h);
        directory.push_back(p);
    }
    allocator.construct(index_to_address(n), std::move(x));
    n += 1;
}

void pop_back() {
    assert(n > 0);
    n -= 1;
    allocator.destroy(index_to_address(n));
    size_type h = directory.size();
    if (h != 1 && population_count(n) == 1) {
        h -= 1;
        allocator.deallocate(directory[h], 1 << h);
        directory.pop_back();
    }
}

void clear() {
    while (size() > 0) {
        pop_back();
    }
}

void swap(pile& other) {
    std::swap(allocator, other_allocator);
    directory.swap(other.directory);
    std::swap(n, other.n);
}

#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>
#include <cstdint>
#include <initializer_list>
#include <memory>
#include "rank-iterator++.h"
#include "resizable_array++.h"
#include <utility>

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;
}
B pointer_allocator;
directory_type directory;
size_type n;

V* index_to_address(size_type rank) const {
  return directory[rank >> shift_amount] + (rank bitand mask);
}

template <typename I>
void append(I first, I past) {
  I p = first;
  I q = first;
  try {
    for (; q != past; ++q) {
      push_back(*q);
    }
  }
  catch (...) {
    for (; p != q; ++p) {
      pop_back();
    }
    throw;
  }
}

void append_n(size_type k, V const& value) {
  size_type i = 0;
  size_type j = 0;
  try {
    for (; j != k; ++j) {
      push_back(value);
    }
  }
  catch (...) {
    for (; i != j; ++i) {
      pop_back();
    }
  }
}

public:

explicit sliced_array(A const& a = A())
  : allocator(a), pointer_allocator(B(a)), directory(B(a)), n(0)
{ }

sliced_array(std::initializer_list<V> s)
  : sliced_array(A())
{ append(s.begin(), s.end()); }
sliced_array(size_type n, A const& a) : sliced_array(a) { append_n(n, V()); }

sliced_array(sliced_array const& other) : allocator(other.allocator), pointer_allocator(other.pointer_allocator), directory(other.pointer_allocator), n(0) {
  directory_type tmp;
  for (size_type i = 0; i < other.directory.size(); ++i) {
    V* p = allocator.allocate(slice_size);
    tmp.push_back(p);
    std::uninitialized_copy(other.directory[i], other.directory[i + slice_size], p);
  }
  directory = std::move(tmp);
  n = other.size();
}

sliced_array& operator=(sliced_array const& other) {
  if (this != &other) {
    sliced_array tmp(other);
    swap(tmp);
  }
  return *this;
}

sliced_array& operator=(sliced_array&& other) {
  if (this != &other) {
    sliced_array tmp(std::move(other));
    swap(tmp);
  }
  return *this;
}

void swap(sliced_array& other) {
  std::swap(allocator, other.allocator);
  std::swap(pointer_allocator, other.pointer_allocator);
  directory.swap(other.directory);
  std::swap(n, other.n);
const_iterator cbegin() const {
    return (size() \neq 0) ? const_iterator(this, 0) : const_iterator(this);
}

const_iterator begin() const {
    return cbegin();
}

iterator begin() {
    return (size() \neq 0) ? iterator(this, 0) : iterator(this);
}

const_iterator cend() const {
    return const_iterator(this);
}

const_iterator end() const {
    return cend();
}

iterator end() {
    return iterator(this);
}

allocator_type get_allocator() const {
    return allocator;
}

size_type size() const {
    return n;
}

void resize(size_type new_size) {
    if (n > new_size) {
        while (n > new_size) {
            pop_back();
        }
        return;
    }
    append_n(new_size - n, V());
}

size_type capacity() const {
    return size_type(-1);
}

void reserve(size_type) {
}
const_reference operator[](size_type i) const {
    return *index_to_address(i);
}

reference operator[](size_type i) {
    return *index_to_address(i);
}

const_reference front() const {
    return *index_to_address(0);
}

reference front() {
    return *index_to_address(0);
}

const_reference back() const {
    return *index_to_address(size() - 1);
}

reference back() {
    return *index_to_address(size() - 1);
}

template <typename... Args>
void emplace_back(Args&&... args) {
    size_type offset = n bitand mask;
    if (offset == 0) {
        V* p = allocator.allocate(slice_size);
        directory.push_back(p);
    } new (index_to_address(n)) V(std::forward<Args>(args)...);
    n += 1;
}

void push_back(V const& x) {
    size_type offset = n bitand mask;
    if (offset == 0) {
        V* p = allocator.allocate(slice_size);
        directory.push_back(p);
    } allocator.construct(directory[n >> shift_amount] + offset, x);
    n += 1;
}

void push_back(V&& x) {
    size_type offset = n bitand mask;
    if (offset == 0) {
        V* p = allocator.allocate(slice_size);
        directory.push_back(p);
    } allocator.construct(directory[n >> shift_amount] + offset, std
move(x));
    n += 1;
}

void pop_back() {
    assert(n > 0);
    n -= 1;
    V* base = directory[n >> shift_amount];
    size_type offset = n bitand mask;
    allocator.destroy(base + offset);
    if (offset == 0) {
        allocator.deallocate(base, slice_size);
        directory.pop_back();
    }
}

void clear() {
    while (n > 0) {
        pop_back();
    }
}

}
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 self_type = fixed_capacity_hat<V, allocator_type>;
using iterator = cphstl::rank_iterator<self_type>;
using const_iterator = cphstl::rank_iterator<self_type const>;

protected:

template <typename H, typename B>
friend class space_efficient_array;

using other_allocator_type = typename A::template rebind<V*>::other;
using directory_type = cphstl::contiguous_array<V*, other_allocator_type>;

allocator_type allocator;
directory_type directory;
size_type n;
size_type m; // a power of two
size_type shift_amount; // lg m
size_type mask;

V* index_to_address(size_type rank) const {
  return directory[rank >> shift_amount] + (rank bitand mask);
}

void append_n(size_type k, V const& value) {
  size_type i = 0;
  size_type j = 0;
  try {
    for ( ; j != k; ++j) {
      push_back(value);
    }
  }
  catch (...) {
    for ( ; i != j; ++i) {
      pop_back();
    }
  }
}

public:

explicit fixed_capacity_hat(A const& a = A())
fixed_capacity_hat(fixed_capacity_hat const& other) :
    allocator(other.allocator), directory(other.allocator), n(0),
    m(0), shift_amount(0), mask(0) {

directory_type tmp;
    tmp.reserve(other.m);
    for (size_type i = 0; i != other.directory.size(); ++i) {
        ∗p = allocator.allocate(m);
        tmp.push_back(p);
        std::uninitialized_copy(other.directory[i], other.directory[i ] + m, p);
    }
    directory = std::move(tmp);
    n = other.n;
    m = other.m;
    shift_amount = other.shift_amount;
    mask = other.mask;
}

fixed_capacity_hat(fixed_capacity_hat&& other) :
    allocator(other.allocator), directory(other.allocator), n(0),
    m(0), shift_amount(0), mask(0) {
    swap(other);
}

~fixed_capacity_hat() {
    clear();
}

fixed_capacity_hat & operator=(fixed_capacity_hat const& other) {
    if (this != &other) {
        fixed_capacity_hat tmp(other);
        swap(tmp);
    }
    return ∗this;
}

fixed_capacity_hat & operator=(fixed_capacity_hat&& other) {
    if (this != &other) {
        fixed_capacity_hat tmp(std::move(other));
        swap(tmp);
    }
    return ∗this;
}

void swap(fixed_capacity_hat& other) {
    std::swap(allocator, other.allocator);
    directory.swap(other.directory);
std::swap(n, other.n);
std::swap(m, other.m);
std::swap(shift_amount, other.shift_amount);
std::swap(mask, other.mask);
}

const_iterator cbegin() const {
    return (size() != 0) ? const_iterator(this, 0) : const_iterator(this);
}

const_iterator begin() const {
    return cbegin();
}

iterator begin() {
    return (size() != 0) ? iterator(this, 0) : iterator(this);
}

const_iterator cend() const {
    return const_iterator(this);
}

const_iterator end() const {
    return cend();
}

iterator end() {
    return iterator(this);
}

allocator_type get_allocator() const {
    return allocator;
}

size_type size() const {
    return n;
}

void resize(size_type new_size) {
    if (n > new_size) {
        while (n > new_size) {
            pop_back();
        }
        return;
    }
    append_n(new_size - n, V());
    n = new_size;
}

size_type capacity() const {
    return m * m;
void reserve(size_type n) {
    assert(m == 0);
    size_type lg_n = ilogb(std::max(size_type(1), n));
    size_type new_m = 1 << (lg_n / 2); // about sqrt of n
    if (new_m * new_m < n) {
        new_m = 2 * new_m;
    }
    directory.reserve(new_m);
    m = new_m;
    shift_amount = ilogb(m);
    mask = m - 1;
}

const_reference operator[](size_type i) const {
    return *index_to_address(i);
}

reference operator[](size_type i) {
    return *index_to_address(i);
}

template<typename... Args>
void emplace_back(Args&&... args) {
    assert(n < capacity());
    size_type offset = n bitand mask;
    if (offset == 0) {
        V* p = allocator.allocate(m);
        directory.push_back(p);
    }
    new (index_to_address(n)) V(std::forward<Args>(args)...);
    n += 1;
}

void push_back(V const& x) {
    assert(n < capacity());
    size_type offset = n bitand mask;
    if (offset == 0) {
        V* p = allocator.allocate(m);
        directory.push_back(p);
    }
    allocator.construct(directory[n >> shift_amount] + offset, x);
    n += 1;
}

void push_back(V& x) {
    assert(n < capacity());
    size_type offset = n bitand mask;
    if (offset == 0) {
        V* p = allocator.allocate(m);
        directory.push_back(p);
    }
35

```cpp
227 } allocator.construct(directory[n >> shift_amount] + offset, std ::move(x));
228 n += 1;
229 }

230 void pop_back() {
231 assert(n != 0);
232 n -= 1;
233 V* base = directory[n >> shift_amount];
234 size_type offset = n bitand mask;
235 allocator.destroy(base + offset);
236 if (offset == 0) {
237 allocator.deallocate(base, m);
238 directory.pop_back();
239 }
240 }
241 }

242 void clear() {
243 while (n > 0) {
244 pop_back();
245 }
246 while (directory.size() > 0) {
247 directory.pop_back();
248 }
249 m = 0;
250 shift_amount = 0;
251 mask = 0;
252 }
253 }
254 }
255 }
256 }

space_efficient_array.h++

/*
This is an experimental implementation of a space-efficient array
that consists of a pool of fixed-capacity hashed array trees; the
directory is a (worst-case-efficient) resizable array.

Author: Jyrki Katajainen @ 2012, 2016
*/

#ifndef __CPHSTL_SPACE_EFFICIENT_ARRAY__
#define __CPHSTL_SPACE_EFFICIENT_ARRAY__
#endif
#include <cassert // assert macro
#include <cstdint // std::size_t, std::ptrdiff_t
#include "fixed_capacity_hat.h++" // cphstl::fixed_capacity_hat
#include <initializer_list // std::initializer_list
#include <memory> // std::allocator
#include "rank-iterator.h++" // cphstl::rank_iterator
#include "resizable_array.h++" // cphstl::resizable_array
```
```cpp
#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__

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

protected:

using H = cphstl::fixed_capacity_hat<V, allocator_type>;
using B = typename A::template rebind<H>::other;
using directory_type = cphstl::resizable_array<H, B>;

allocator_type allocator;
directory_type directory;
size_type n;

V* index_to_address(size_type rank) const {
    if (rank < 2) {
        return directory[0].index_to_address(rank);
    }
    std::size_t h = whole_number_logarithm(rank);
    size_type offset = rank - (1 << h);
    return directory[h].index_to_address(offset);
}

template<typename S, typename I>
void append(S& sequence, I some, I past) {
    I p = some;
    I q = some;
    try {
        for (; q != past; ++q) {
            sequence.push_back(*q);
        }
    } catch (...) {
        for (; p != q; ++p) {
            sequence.pop_back();
        }
        throw;
    }
}

void append_n(size_type k, V const& value) {
    size_type i = 0;
    size_type j = 0;
    try {
        for (; j != k; ++j) {
            push_back(value);
        }
    } catch (...) {
        for (; i != j; ++i) {
            pop_back();
        }
    }
public:

explicit space_efficient_array(A const& a = A())
  : allocator(a), directory(a), n(0) {
    directory.push_back(std::move(H(allocator)));
    directory[0].reserve(2);
  }

space_efficient_array(std::initializer_list<V> s)
  : space_efficient_array(A()) {
    append(*this, s.begin(), s.end());
  }

space_efficient_array(space_efficient_array const& other)
  : allocator(other allocator), directory(other allocator), n(0)
  {
    space_efficient_array tmp;
    append(tmp, other.begin(), other.end());
    (*this).swap(tmp);
  }

space_efficient_array(space_efficient_array&& other)
  : allocator(other allocator), directory(other allocator), n(0)
  {
    (*this).swap(other);
  }

∼space_efficient_array() {
  clear();
  directory.clear();
}

space_efficient_array& operator=(space_efficient_array const& other) {
  if (this ≠ &other) {
    space_efficient_array tmp(other);
    (*this).swap(tmp);
  }
  return *this;
}

space_efficient_array& operator=(space_efficient_array&& other) {
  if (this ≠ &other) {
    space_efficient_array tmp(std::move(other));
    (*this).swap(tmp);
  }
  return *this;
}
void swap(space_efficient_array& other) {
    std::swap(allocation, other.allocation);
    directory.swap(other.directory);
    std::swap(n, other.n);
}

const_iterator cbegin() const {
    return (size() != 0) ? const_iterator(this, 0) : const_iterator(this);
}

const_iterator begin() const {
    return cbegin();
}

iterator begin() {
    return (size() != 0) ? iterator(this, 0) : iterator(this);
}

const_iterator cend() const {
    return const_iterator(this);
}

const_iterator end() const {
    return cend();
}

iterator end() {
    return iterator(this);
}

allocator_type get_allocator() const {
    return allocation;
}

size_type size() const {
    return n;
}

void resize(size_type new_size) {
    if (n > new_size) {
        while (n > new_size) {
            pop_back();
        }
        return;
    }

    append_n(new_size - n, V());
}

size_type capacity() const {
    return size_type(-1);
}
void reserve(size_type) {
}

const_reference operator[](size_type i) const {
    return *index_to_address(i);
}

reference operator[](size_type i) {
    return *index_to_address(i);
}

const_reference front() const {
    return *index_to_address(0);
}

reference front() {
    return *index_to_address(0);
}

const_reference back() const {
    return *index_to_address(size() - 1);
}

reference back() {
    return *index_to_address(size() - 1);
}

template<typename... Args>
void emplace_back(Args&&... args) {
    size_type h = directory.size();
    if (n > 1 and population_count(n) == 1) {
        directory.push_back(std::move(H(allocator)));
        directory[h].reserve(1 << h);
        h += 1;
    }
    directory[h - 1].emplace_back(std::forward<Args>(args)...);
    n += 1;
}

void push_back(V const& x) {
    size_type h = directory.size();
    if (n > 1 and population_count(n) == 1) {
        directory.push_back(std::move(H(allocator)));
        directory[h].reserve(1 << h);
        h += 1;
    }
    directory[h - 1].push_back(x);
    n += 1;
}

void push_back(V&& x) {
size_type h = directory.size();
if (n > 1 and population_count(n) == 1) {
    directory.push_back(std::move(H(allocator)));
    directory[h].reserve(1 << h);
    h += 1;
}
directory[h - 1].push_back(std::move(x));
n += 1;
}
void pop_back() {
    assert(n != 0);
    n -= 1;
    size_type h = directory.size();
    directory[h - 1].pop_back();
    if (h != 1 and population_count(n) == 1) {
        directory[h - 1].clear();
        directory.pop_back();
    }
}
void clear() {
    while (size() > 0) {
        pop_back();
    }
}
}
#endif

Containers

leda-array.h++

/*
The interface of cphleda::array

Authors: Jyrki Katajainen, Bo Simonsen © 2009, 2016
*/
#include <algorithm> // std::unique, std::sort, std::random_shuffle, std::lower_bound, std::upper_bound
#include <cassert> // assert macro
#include "leda-compare-functions.i++"
#include <memory> // std::allocator
#include <vector> // std::vector
#include <sstream> // std::stringstream
#include <utility> // std::swap, std::pair
namespace cphleda {

template<typename V, typename K = std::vector<V, std::allocator<V>>>

class array {
public:
    using value_type = V;
    using kernel_type = K;
    using iterator = typename K::iterator;
    using const_iterator = typename K::const_iterator;
    using item = iterator;
    using size_type = int;

    array(size_type, size_type);
    array(size_type);
    array();
    array(size_type, V, V);
    array(size_type, V, V, V);
    array(size_type, V, V, V, V);
    array(array const&);

    array& operator=(array const&);

    item first_item();
    item last_item();
    item next_item(item);
    item prev_item(item);

    V& inf(item);

    iterator begin();
    iterator end();
    const_iterator begin() const;
    const_iterator end() const;

    V& get(size_type);
    V const& get(size_type) const;

    void set(size_type, V);
    V& operator[](size_type);

    void copy(size_type, array const&, size_type);
    void copy(size_type, size_type);
    void resize(size_type, size_type);
    void resize(size_type);

    size_type low() const;
    size_type high() const;
    size_type size() const;

    void init(V);
    bool C_style() const;
void swap(size_type, size_type);
void sort(size_type (*)(V const&, V const&));
void sort(size_type, size_type);
void sort();
void sort(size_type (*)(V const&, V const&, size_type, size_type);
void permute();
void permute(size_type, size_type);
size_type binary_search(size_type (*)(V const&, V const&), V);
size_type binary_search(V);
size_type binary_locate(size_type (*)(V const&, V const&), V);
size_type binary_locate(V);
size_type unique();
void print(std::ostream&, char = '\n');
void print(char = '\n');
void print(char const*, char = '\n');
void read(std::istream&);
void read();
void read(char const*);

protected:
void fill(K&, size_type, size_type);
template <typename S, typename I>
void append(S&, I, I);
std::pair<iterator, iterator> get_iterators(size_type, size_type)
;

private:
size_type s;
size_type e;
kernl_type kernel;

};

template <typename V, typename K>
std::istream & operator>>(std::istream&, array<V, K>&);

#include "leda-array.i++" // implementation
/* An implementation of cphleda::array
Authors: Jyrki Katajainen, Bo Simonsen © 2009, 2016 */
	namespace cphleda {

template<typename V, typename K>
array<V, K>::array(size_type a, size_type b) :
s(0), e(-1), kernel() {
fill(kernel, 0, b - a);
s = a;
e = b;
}

template<typename V, typename K>
array<V, K>::array(size_type n) :
s(0), e(-1), kernel() {
fill(kernel, 0, n);
e = n - 1;
}

template<typename V, typename K>
array<V, K>::array() :
s(0), e(-1), kernel() {
}

template<typename V, typename K>
array<V, K>::array(size_type low, V x, V y) :
s(0), e(-1), kernel() {
std::initializer_list<V> list = {x, y};
append(kernel, list.begin(), list.end());
s = low;
e = low + 1;
}

template<typename V, typename K>
array<V, K>::array(size_type low, V x, V y, V z) :
s(0), e(-1), kernel() {
std::initializer_list<V> list = {x, y, z};
append(kernel, list.begin(), list.end());
s = low;
e = low + 2;
}

template<typename V, typename K>
array<V, K>::array(size_type low, V x, V y, V z, V w) :
s(0), e(-1), kernel() {
std::initializer_list<V> list = {x, y, z, w};
append(kernel, list.begin(), list.end());
s = low;
e = low + 3;
}

template<typename V, typename K>
array<V, K>::array(array const& a) :
s(0), e(-1), kernel() {
K tmp;
append(tmp, a.begin(), a.end());
kernel.swap(tmp);
s = a.low();
e = a.high();
}

template<typename V, typename K>
array<V, K>& array<V, K>::operator=(array const& other) {
if (this != &other) {
K tmp;
append(tmp, other.begin(), other.end());
kern. swap(tmp);
s = other.low();
e = other.high();
}
return *this;
}

template<typename V, typename K>
V& array<V, K>::get(size_type i) {
assert(low() ≤ i and i ≤ high());
return kernel[i - low()];
}

template<typename V, typename K>
V const& array<V, K>::get(size_type i) const {
assert(low() ≤ i and i ≤ high());
return kernel[i - low()];
}

template<typename V, typename K>
void array<V, K>::set(size_type i, V e) {
assert(low() ≤ i and i ≤ high());
get(i) = e;
}

template<typename V, typename K>
V& array<V, K>::operator[](size_type i) {
#if not defined(LEDA_CHECKING_OFF)
if (i < low() or i > high()) {

```
throw std::out_of_range("ERROR array: index out of range");

#endif

return get(i);

}  

template<typename V, typename K>
void array<V, K>::copy(size_type x, array<V, K> const& other,  
  size_type y) {
  assert(low() ≤ x and x ≤ high());
  assert(other.low() ≤ y and y ≤ other.high());
  set(x, other.get(y));
}

template<typename V, typename K>
void array<V, K>::copy(size_type x, size_type y) {
  assert(low() ≤ x and x ≤ high());
  assert(low() ≤ y and y ≤ high());
  set(x, get(y));
}

template<typename V, typename K>
void array<V, K>::resize(size_type a, size_type b) {
  assert(a ≤ b);
  K tmp;
  if (e < a) {
    fill(tmp, 0, b - a + 1);
  } else if (e ≤ b) {
    if (s < a) {
      append(tmp, begin() + (a - s), end());
      fill(tmp, e - a + 1, b - a + 1);
    } else {
      fill(tmp, 0, s - a);
      append(tmp, begin(), end());
      fill(tmp, tmp.size(), b - a + 1);
    }
  } else if (s < a) {
    append(tmp, kernel.begin() + (a - s), kernel.begin() + (b - s + 1));
  } else if (s ≤ b) {
    fill(tmp, 0, s - a);
    append(tmp, kernel.begin(), kernel.begin() + (b - s + 1));
  } else {
    assert(s > b);
    fill(tmp, 0, b - a + 1);  

kernel.swap(tmp);
s = a;
e = b;
}

template <typename V, typename K>
void array<V, K>::resize(size_type n) {
    resize(0, n - 1);
}

template <typename V, typename K>
typename array<V, K>::size_type array<V, K>::low() const {
    return s;
}

template <typename V, typename K>
typename array<V, K>::size_type array<V, K>::high() const {
    return e;
}

template <typename V, typename K>
typename array<V, K>::size_type array<V, K>::size() const {
    return e - s + 1;
}

template <typename V, typename K>
void array<V, K>::init(V x) {
    for (size_type i = low(); i <= high(); ++i) {
        set(i, x);
    }
}

template <typename V, typename K>
bool array<V, K>::C_style() const {
    return low() == 0;
}

template <typename V, typename K>
void array<V, K>::swap(size_type i, size_type j) {
    std::swap(get(i), get(j));
}

template <typename V, typename K>
typename array<V, K>::size_type array<V, K>::unique() {
    std::pair<iterator, iterator> p = get_iterators(low(), high());
    iterator new_end = std::unique(p.first, p.second);
    size_type i = (new_end - p.first) + low();
    return i - 1;
}

/* sort functions */
template<typename V, typename K>
void array<V, K>::sort(size_type (*cmp)(V const&, V const&)) {
    sort(cmp, low(), high());
}

template<typename V, typename K>
void array<V, K>::sort(size_type l, size_type h) {
    sort(compare, l, h);
}

template<typename V, typename K>
void array<V, K>::sort() {
    sort(compare);
}

template<typename V, typename K>
void array<V, K>::permute(size_type (*cmp)(V const&, V const&),
    size_type l, size_type h) {
    std::pair<iterator, iterator> p = get_iterators(l, h);
    std::sort(p.first, p.second, stl_compare_less<V>(cmp));
}

/* permute functions */

template<typename V, typename K>
void array<V, K>::permute() {
    permute(low(), high());
}

template<typename V, typename K>
void array<V, K>::permute(size_type l, size_type h) {
    std::pair<iterator, iterator> p = get_iterators(l, h);
    std::random_shuffle(p.first, p.second);
}

/* binary-search functions */

template<typename V, typename K>
typename array<V, K>::size_type
array<V, K>::binary_search(size_type (*cmp)(V const&, V const&),
    V x) {
    std::pair<iterator, iterator> p = get_iterators(low(), high());
    iterator it = std::lower_bound(p.first, p.second, x,
        stl_compare_less<V>(cmp));
    if (it == p.second) {
        return low() - 1;
    }
    return (it - p.first) + low();
}

template<typename V, typename K>
template<typename V, typename K>
typearray<V, K>::size_type array<V, K>::binary_search(V x) {
    return binary_search(compare, x);
}

template<typename V, typename K>
typearray<V, K>::size_type array<V, K>::binary_locate(size_type (*cmp)(V const&, V const&), V x) {
    std::pair<iterator, iterator> p = get_iterators(low(), high());
    iterator it = std::upper_bound(p.first, p.second, x,
        stl_compare_less<V>(cmp));
    if (it == p.second) {
        return low() - 1;
    }
    size_type i = (it - p.first) - 1 + low();
    if (cmp(x, get(i)) > 0) {
        return low() - 1;
    }
    return i;
}

/* print functions */

template<typename V, typename K>
void array<V, K>::print(std::ostream& o, char space) {
    std::stringstream sss;
    sss << space;
    for(size_type i = low(); i <= high(); ++i) {
        std::stringstream ss;
        ss << get(i);
        o << ss.str();
        if(i != high()) {
            o << space;
        }
    }
}

template<typename V, typename K>
void array<V, K>::print(char const* header, char space) {
    print(std::cout, space);
}

template<typename V, typename K>
void array<V, K>::print(char const* header, char space) {
    std::cout << header << std::endl;
    print(std::cout, space);
}
template <typename V, typename K>
void array<V, K>::read(std::istream& s) {
    V v;
    for(size_type i = low(); i <= high(); ++i) {
        s >> v;
        set(i, v);
    }
}

template <typename V, typename K>
void array<V, K>::read() {
    read(std::cin);
}

template <typename V, typename K>
void array<V, K>::read(const char* header) {
    std::cout << header;
    read();
}

/* helpers */

template <typename V, typename K>
void array<V, K>::fill(K& data, size_type j, size_type n) {
    size_type k = j;
    try {
        for ( ; k != n; ++k) {
            data.push_back(V());
        }
    } catch (...) {
        for ( ; j != k; ++j) {
            data.pop_back();
        }
        throw;
    }
}

template <typename V, typename K>
template <typename S, typename I>
void array<V, K>::append(S& sequence, I some, I past) {
    I p = some;
    I q = some;
    try {
        for ( ; q != past; ++q) {
            sequence.push_back(*q);
        }
    } catch (...) {
        }
for ( ; p != q; ++p) {
    sequence.pop_back();
}
throw;
}

template <typename V, typename K>
std::pair<typename array<V, K>::iterator, typename array<V, K>::iterator>
array<V, K>::get_iterators(size_type l, size_type h) {
    size_type start_index = l - low();
    size_type end_index = h - low();
    iterator begin_it = kernel.begin();
    iterator end_it = kernel.begin();
    begin_it += start_index;
    end_it += end_index + 1;
    return std::pair<iterator, iterator>(begin_it, end_it);
}

// we use iterators to implement next_item, prev_item, and inf */

template <typename V, typename K>
typename array<V, K>::item array<V, K>::first_item() {
    return kernel.begin();
}

template <typename V, typename K>
typename array<V, K>::item array<V, K>::last_item() {
    return prev_item(kernel.end());
}

template <typename V, typename K>
typename array<V, K>::item array<V, K>::next_item(item a) {
    iterator p(a);
    ++p;
    return p;
}

template <typename V, typename K>
typename array<V, K>::item array<V, K>::prev_item(item a) {
    iterator p(a);
    --p;
    return p;
}

template <typename V, typename K>
V& array<V, K>::inf(item a) {
    iterator p(a);
    return &p;
}
According to the C++ standard [2014], a vector is a sequence that supports random-access iterators. More precisely, it satisfies all of the requirements of a container, of a reversible container, and of a sequence container, including most of the optional sequence container requirements, of an allocator-aware container, and, for an element type other than bool, of a contiguous container. The exceptions are the push_front, pop_front, and emplace_front member functions, which are not provided.
#ifndef __CPHSTL_VECTOR__
#define __CPHSTL_VECTOR__

#include <algorithm> // std::equal, std::lexicographical_compare, std::rotate
#include <cstdlib> // std::size_t, std::ptrdiff_t
#include <iterator> // std::reverse_iterator
#include <limits> // std::numeric_limits
#include <memory> // std::allocator, std::allocator_traits
#include <stdexcept> // std::length_error, std::out_of_range
#include <type_traits> // std::conditional, std::true_type, std::false_type
#include <vector> // std::vector
#include <utility> // std::swap, std::move, std::forward

namespace cphstl {

    template <typename V, typename A = std::allocator<V>,
              typename K = std::vector<V, A>>
    class vector {

    public:

        // types

        using value_type = V;
        using allocator_type = A;
        using reference = typename K::reference;
        using const_reference = typename K::const_reference;
        using pointer = typename std::allocator_traits<A>::pointer;
        using const_pointer = typename std::allocator_traits<A>::const_pointer;
        using size_type = std::size_t;
        using difference_type = std::ptrdiff_t;
        using iterator = typename K::iterator;
        using const_iterator = typename K::const_iterator;
        using reverse_iterator = std::reverse_iterator<iterator>;
        using const_reverse_iterator = std::reverse_iterator<const_iterator>;

    public:

        // structs

        vector() noexcept;
        explicit vector(A const&) noexcept;
        explicit vector(size_type, A const& = A());
        vector(size_type, V const& , A const& = A());

        template <typename I>
        vector(I, I, A const& = A());

    };
vector(vector const&);
vector(vector const&, A const&);
vector(vector&&) noexcept;
vector(vector&&, A const&) noexcept(
    std::allocator_traits<A>::
    propagate_on_container_move_assignment::value);
// C++17 or std::allocator_traits<A>::is_always_equal::value);
vector(std::initializer_list<V>, A const& = A());
~vector();
vector& operator=(vector const&);
vector& operator=(vector&&) noexcept(
    std::allocator_traits<A>::
    propagate_on_container_move_assignment::value);
// C++17 or std::allocator_traits<A>::is_always_equal::value);
vector& operator=(std::initializer_list<V>);

// iterators

iterator begin() noexcept;
const_iterator begin() const noexcept;
iterator end() noexcept;
const_iterator end() const noexcept;
reverse_iterator rbegin() noexcept;
const_reverse_iterator rbegin() const noexcept;
reverse_iterator rend() noexcept;
const_reverse_iterator rend() const noexcept;

const_iterator cbegin() const noexcept;
const_iterator cend() const noexcept;
const_reverse_iterator crbegin() const noexcept;
const_reverse_iterator crend() const noexcept;

// accessors

A get_allocator() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;
size_type capacity() const noexcept;
bool empty() const noexcept;

const_reference operator[](size_type) const;
const_reference at(size_type) const;
const_reference front() const;
const_reference back() const;

// modifiers


reference operator[](size_type);
reference at(size_type);
reference front();
reference back();

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

void resize(size_type);
void resize(size_type, V const&);
void reserve(size_type);
void shrink_to_fit();

template <typename... Args>
void emplace_back(Args&&...);

void push_back(V const&);
void push_back(V&&);
void pop_back();

template <typename... Args>
iterator emplace(iterator, Args&&...);

iterator insert(iterator, V const&);
iterator insert(iterator, V&&);
iterator insert(iterator, size_type, V const&);

template <typename I>
iterator insert(iterator, I, I);

iterator insert(iterator, std::initializer_list<V>);
iterator erase(iterator);
iterator erase(iterator, iterator);
void swap(vector&) noexcept(
    std::allocator_traits<A>::propagate_on_container_swap::value);
// C++17 or std::allocator_traits<A>::is_always_equal::value);
void clear() noexcept;

protected:

    // helpers

template <typename S, typename I>
void append(S&, I, I);

template <typename S>
void append_n(S&, size_type, V const&);

template <typename I>
void assign_dispatch(I, I, std::false_type);

template <typename I>
void assign_dispatch(I, I, std::true_type);
void shrink_to_fit_dispatch(std::false_type);
void shrink_to_fit_dispatch(std::true_type);

template<typename I>
iterator insert_dispatch(iterator, I, I, std::true_type);

template<typename I>
iterator insert_dispatch(iterator, I, I, std::false_type);

using kernel_type = K;
kernel_type kernel;
};

template<typename V, typename A, typename K>
bool operator==(vector<V, A, K> const&, vector<V, A, K> const&);

template<typename V, typename A, typename K>
bool operator<(vector<V, A, K> const&, vector<V, A, K> const&);

template<typename V, typename A, typename K>
bool operator!=(vector<V, A, K> const&, vector<V, A, K> const&);

template<typename V, typename A, typename K>
bool operator>(vector<V, A, K> const&, vector<V, A, K> const&);

#include "stl-vector.i++" // implementation

A vector is a container that delegates its work to the given kernel
namespace cphstl {

    // helpers

    template<typename V, typename A, typename K>
    template<typename S, typename I>
    void vector<V, A, K>::append(S& sequence, I some, I past) {
        I p = some;
        I q = some;
        try {
            for (; q != past; ++q) {
                sequence.push_back(*q);
            }
        } catch (...) {
            for (; p != q; ++p) {
                sequence.pop_back();
            }
            throw;
        }
    }

    template<typename V, typename A, typename K>
    template<typename S>
    void vector<V, A, K>::append_n(S& sequence, size_type k, V const& value) {
        size_type i = 0;
        size_type j = 0;
        try {
            for (; j != k; ++j) {
                sequence.push_back(value);
            }
        } catch (...) {
            for (; i != j; ++i) {
                sequence.pop_back();
            }
        }
    }

    // default constructor

    template<typename V, typename A, typename K>
    vector<V, A, K>::vector() noexcept : kernel(A()) {
    }

    // explicit constructors
template <typename V, typename A, typename K>
vector<V, A, K>::vector(A const& a) noexcept
: kernel(a) {}

template <typename V, typename A, typename K>
vector<V, A, K>::vector(size_type n, A const& a)
: kernel(a) {
    append_n(kernel, n, V());
}

// parametrized constructors

template <typename V, typename A, typename K>
vector<V, A, K>::vector(size_type n, V const& v, A const& a)
: kernel(a) {
    append_n(kernel, n, v);
}

template <typename V, typename A, typename K>
template <typename I>
vector<V, A, K>::vector(I p, I q, A const& a)
: kernel(a) {
    append(kernel, p, q);
}

// copy constructors

template <typename V, typename A, typename K>
vector<V, A, K>::vector(vector const& other)
: kernel(other.get_allocator()) {
K tmp(other.get_allocator());
append(tmp, other.begin(), other.end());
kernel.swap(tmp);
}

template <typename V, typename A, typename K>
vector<V, A, K>::vector(vector const& other, A const& a)
: kernel(a) {
K tmp(a);
append(tmp, other.begin(), other.end());
kernel.swap(tmp);
}

// move constructors

template <typename V, typename A, typename K>
vector<V, A, K>::vector(vector&& other) noexcept
: kernel(other.get_allocator()) {
kernel.swap(other.kernel);
}
template <typename V, typename A, typename K>
vector<V, A, K>::vector(std::initializer_list<V> other, A const& a)
    : kernel(a) {
    K tmp(a);
    append(tmp, other.begin(), other.end());
    kernel.swap(tmp);
}

// destructor

template <typename V, typename A, typename K>
vector<V, A, K>::~vector() {
    clear();
}

// copy assignment

template <typename V, typename A, typename K>
vector<V, A, K>& vector<V, A, K>::operator=(vector const& other) {
    if (this != &other) {
        K tmp(other.kernel);
        kernel.swap(tmp);
    }
    return *this;
}

// move assignment

template <typename V, typename A, typename K>
vector<V, A, K>& vector<V, A, K>::operator=(vector&& other) noexcept(
    std::allocator_traits<A>::propagate_on_container_move_assignment::value) {
    // C++17 or std::allocator_traits<A>::is_always_equal::value)
    if (this != &other) {
        clear();
        kernel.swap(other.kernel);
    }
    return *this;
template <typename V, typename A, typename K>
vector<V, A, K>& vector<V, A, K>::operator=(std::initializer_list<V> other) {
    assign_dispatch(other.begin(), other.end(), std::false_type());
    return *this;
}

// assign

template <typename V, typename A, typename K>
void vector<V, A, K>::assign(size_type n, V const& v) {
    K tmp(kernel.get_allocator());
    append_n(tmp, n, v);
    kernel.swap(tmp);
}

template <typename V, typename A, typename K>
void vector<V, A, K>::assign(I p, I q) {
    assign_dispatch(p, q, typename std::conditional<std::is_integral<I>::value, std::true_type, std::false_type>::type());
}

// begin

template <typename V, typename A, typename K>
void vector<V, A, K>::assign_dispatch(I p, I q, std::false_type) {
    K tmp(kernel.get_allocator());
    append(tmp, p, q);
    kernel.swap(tmp);
}
template<typename V, typename A, typename K>
typename vector<V, A, K>::iterator vector<V, A, K>::begin() noexcept {
    return kernel.begin();
}

template<typename V, typename A, typename K>
typename vector<V, A, K>::begin() noexcept {
    return kernel.begin();
}

// end

template<typename V, typename A, typename K>
typename vector<V, A, K>::const_iterator vector<V, A, K>::begin() const noexcept {
    return const_iterator(kernel.begin());
}

template<typename V, typename A, typename K>
typename vector<V, A, K>::const_iterator vector<V, A, K>::begin() const noexcept {
    return const_iterator(kernel.begin());
}

// rbegin

template<typename V, typename A, typename K>
typename vector<V, A, K>::reverse_iterator vector<V, A, K>::rbegin() noexcept {
    return reverse_iterator(end());
}

template<typename V, typename A, typename K>
typename vector<V, A, K>::reverse_iterator vector<V, A, K>::rbegin() const noexcept {
    return reverse_iterator(end());
}

// rend

template<typename V, typename A, typename K>
typename vector<V, A, K>::reverse_iterator vector<V, A, K>::rend() noexcept {
    return reverse_iterator(begin());
}

template<typename V, typename A, typename K>
typename vector<V, A, K>::reverse_iterator vector<V, A, K>::rend() const noexcept {
    return reverse_iterator(begin());
}
// cbegin

template <typename V, typename A, typename K>
typename vector<V, A, K>::const_iterator vector<V, A, K>::cbegin()
    const noexcept {
    return const_iterator(kernel.begin());
}

// cend

template <typename V, typename A, typename K>
typename vector<V, A, K>::const_iterator vector<V, A, K>::cend()
    const noexcept {
    return const_iterator(kernel.end());
}

// crbegin

template <typename V, typename A, typename K>
typename vector<V, A, K>::const_reverse_iterator vector<V, A, K>::crbegin()
    const noexcept {
    return const_reverse_iterator(end());
}

// crend

template <typename V, typename A, typename K>
typename vector<V, A, K>::const_reverse_iterator vector<V, A, K>::crend()
    const noexcept {
    return const_reverse_iterator(begin());
}

// get_allocator

template <typename V, typename A, typename K>
A vector<V, A, K>::get_allocator() const noexcept {
    return kernel.get_allocator();
}

// size

template <typename V, typename A, typename K>
typename vector<V, A, K>::size_type vector<V, A, K>::size()
    const noexcept {
    return kernel.size();
}

// max_size

template <typename V, typename A, typename K>
typename vector<V, A, K>::size_type vector<V, A, K>::max_size()
const noexcept {
    return std::numeric_limits<difference_type>::max();
}

// capacity

template <typename V, typename A, typename K>
typename vector<V, A, K>::size_type vector<V, A, K>::capacity()
    const noexcept {
    return kernel.capacity();
}

// empty

template <typename V, typename A, typename K>
bool vector<V, A, K>::empty() const noexcept {
    return size() == size_type(0);
}

// operator []

template <typename V, typename A, typename K>
typename vector<V, A, K>::reference vector<V, A, K>::operator[](size_type i) {
    return reference(kernel.operator[](i));
}

template <typename V, typename A, typename K>
typename vector<V, A, K>::const_reference vector<V, A, K>::operator[](size_type i) const {
    return const_reference(kernel.operator[](i));
}

// at

template <typename V, typename A, typename K>
typename vector<V, A, K>::reference vector<V, A, K>::at(size_type i) {
    if (i >= size()) {
        throw std::out_of_range("index out of bounds");
    }
    return reference(operator[](i));
}

template <typename V, typename A, typename K>
typename vector<V, A, K>::const_reference vector<V, A, K>::at(size_type i) const {
    if (i >= size()) {
        throw std::out_of_range("index out of bounds");
    }
    return const_reference(operator[](i));
}
// front

template <typename V, typename A, typename K>
typename vector<V, A, K>::reference vector<V, A, K>::front() {
    iterator first = kernel.begin();
    return reference(*first);
}

template <typename V, typename A, typename K>
typename vector<V, A, K>::const_reference vector<V, A, K>::front() const {
    const_iterator first = kernel.begin();
    return const_reference(*first);
}

// back

template <typename V, typename A, typename K>
typename vector<V, A, K>::reference vector<V, A, K>::back() {
    iterator const last = --end();
    return reference(*last);
}

template <typename V, typename A, typename K>
typename vector<V, A, K>::const_reference vector<V, A, K>::back() const {
    const_iterator const last = --end();
    return const_reference(*last);
}

// data

template <typename V, typename A, typename K>
V* vector<V, A, K>::data() noexcept {
    return kernel.data();
}

template <typename V, typename A, typename K>
V const* vector<V, A, K>::data() const noexcept {
    return kernel.data();
}

// resize

template <typename V, typename A, typename K>
void vector<V, A, K>::resize(size_type n) {
    resize(n, V());
}

template <typename V, typename A, typename K>
void vector<V, A, K>::resize(size_type n, V const& v) {

if (size() > n) {
    while (size() > n) {
        pop_back();
    }
    return;
}
append_n(kernel, n - size(), v);
}

// reserve

template <typename V, typename A, typename K>
void vector<V, A, K>::reserve(size_type s) {
    kernel.reserve(s);
}

// shrink_to_fit

template <typename T>
class has_shrink_to_fit {
    template <typename U, void (U::*)(void)> struct check;
    template <typename U> static char member(check<U, &U::shrink_to_fit>{});
    template <typename U> static int member(...);

    public:
        enum { value = sizeof(member<T>(nullptr)) == sizeof(char) };
    }

template <typename V, typename A, typename K>
void vector<V, A, K>::shrink_to_fit() {
    shrink_to_fit_dispatch<typename std::conditional<
        has_shrink_to_fit<K>::value, std::true_type, std::false_type
    >::type());
}

template <typename V, typename A, typename K>
void vector<V, A, K>::shrink_to_fit_dispatch(std::true_type) {
    kernel.shrink_to_fit();
}

template <typename V, typename A, typename K>
void vector<V, A, K>::shrink_to_fit_dispatch(std::false_type) {
    // do nothing
}
// emplace_back

template <typename V, typename A, typename K>
template <typename... Args>
void vector<V, A, K>::emplace_back(Args&&... args) {
    kernel.emplace_back(std::forward<Args>(args)...);
}

// push_back

template <typename V, typename A, typename K>
void vector<V, A, K>::push_back(V const& v) {
    kernel.push_back(v);
}

template <typename V, typename A, typename K>
void vector<V, A, K>::push_back(V&& v) {
    kernel.push_back(std::move(v));
}

// pop_back

template <typename V, typename A, typename K>
void vector<V, A, K>::pop_back() {
    kernel.pop_back();
}

// emplace

template <typename V, typename A, typename K>
template <typename... Args>
type_vector<V, A, K>::iterator vector<V, A, K>::emplace(
    iterator p, Args&&... args) {
    size_type i = p - begin();
    size_type j = size();
    emplace_back(std::forward<Args>(args)...);
    std::rotate(begin() + i, begin() + j, end());
    return begin() + i;
}

// single-element insert

template <typename V, typename A, typename K>
type_vector<V, A, K>::iterator vector<V, A, K>::insert(iterator
    p, V const& v) {
    size_type i = p - begin();
    size_type j = size();
    push_back(v);
    std::rotate(begin() + i, begin() + j, end());
    return begin() + i;
}
template <typename V, typename A, typename K>

typename vector<V, A, K>::iterator vector<V, A, K>::insert(iterator p, V&& v) {
    size_type i = p - begin();
    size_type j = size();
    push_back(std::move(v));
    std::rotate(begin() + i, begin() + j, end());
    return begin() + i;
}

// multiple-element inserts

template <typename V, typename A, typename K>

type_name vector<V, A, K>::iterator vector<V, A, K>::insert(iterator p, size_type n, V const& v) {
    return insert_fill(p, n, v);
}

template <typename V, typename A, typename K>

template<typename I>

type_name vector<V, A, K>::iterator vector<V, A, K>::insert_dispatch(iterator p, I q, I r) {
    return insert_dispatch(p, q, r, typename std::conditional<std::is_integral<I>::value, std::true_type, std::false_type>::type());
}

template <typename V, typename A, typename K>

template<typename I>

type_name vector<V, A, K>::iterator vector<V, A, K>::insert_dispatch(iterator p, I q, I r, std::true_type) {
    return insert_fill(p, static_cast<size_type>(n), static_cast<V>(v));
}

template <typename V, typename A, typename K>

template<typename I>

type_name vector<V, A, K>::iterator vector<V, A, K>::insert_dispatch(iterator p, I q, I r, std::false_type) {
    return insert_range(p, q, r);
}

template <typename V, typename A, typename K>

type_name vector<V, A, K>::iterator vector<V, A, K>::insert_dispatch(iterator p, std::initializer_list<V> l) {
    return insert_range(p, l, begin(), l, end());
}

template <typename V, typename A, typename K>

type_name vector<V, A, K>::iterator vector<V, A, K>::insert_fill(iterator p, size_type n, V const& v) {
    size_type i = p - begin();
size_type j = size();
append_n(kernel, n, v);
std::rotate(begin() + i, begin() + j, end());
return begin() + i;
}

template <typename V, typename A, typename K>
type name vector<V, A, K>::iterator vector<V, A, K>::insert_range(
    iterator p, I q, I r) {
size_type i = p - begin();
size_type j = size();
append(kernel, q, r);
std::rotate(begin() + i, begin() + j, end());
return begin() + i;
}

// single-element erase

template <typename V, typename A, typename K>
type name vector<V, A, K>::iterator vector<V, A, K>::erase(iterator p) {
assert(p != end());
size_type i = p - begin();
std::rotate(begin() + i, begin() + i + 1, end());
pop_back();
return begin() + i;
}

// multiple-element erase

template <typename V, typename A, typename K>
type name vector<V, A, K>::iterator vector<V, A, K>::erase(iterator p, iterator q) {
size_type i = p - begin();
size_type j = q - begin();
std::rotate(begin() + i, begin() + j, end());
while (i != j) {
pop_back();
++i;
}
return begin() + i;
}

// clear

template <typename V, typename A, typename K>
void vector<V, A, K>::clear() noexcept {
while (!empty()) {
pop_back();
}
}
// swap

template<typename V, typename A, typename K>
void vector<V, A, K>::swap(vector<V, A, K>& other) noexcept(
    std::allocator_traits<A>::propagate_on_container_swap::value) {
    // C++17 or std::allocator_traits<A>::is_always_equal::value
    kernel.swap(other.kernel);
}

template<typename V, typename A, typename K>
void swap(vector<V, A, K>& x, vector<V, A, K>& y) noexcept(
    noexcept(x.swap(y))) {
    x.swap(y);
}

// operator ==

template<typename V, typename A, typename K>
bool operator==(vector<V, A, K> const& r, vector<V, A, K> const& s) {
    return (r.size() == s.size() and std::equal(r.begin(), r.end(), s.begin()));
}

// operator <

template<typename V, typename A, typename K>
bool operator<(vector<V, A, K> const& r, vector<V, A, K> const& s) {
    return std::lexicographical_compare(r.begin(), r.end(), s.begin() , s.end());
}

// operator ≠

template<typename V, typename A, typename K>
bool operator!=(vector<V, A, K> const& r, vector<V, A, K> const& s) {
    return not (r == s);
}

// operator >

template<typename V, typename A, typename K>
bool operator>(vector<V, A, K> const& r, vector<V, A, K> const& s) {
    return (s < r);
}

// operator ≥
template <typename V, typename A, typename K>
bool operator>(vector<V, A, K> const& r, vector<V, A, K> const& s) {
    return not (r < s);
}

// operator ≤

template <typename V, typename A, typename K>
bool operator<=(vector<V, A, K> const& r, vector<V, A, K> const& s) {
    return not (s < r);
}

Helpers

**rank-iterator.h++**

```cpp
/*
 * A rank iterator encapsulates a location of an element by storing a
 * (pointer, rank) pair; the pointer refers to a data structure,
 * called the owner, that contains the element referred to and the
 * rank is the index of that element within the data structure.
 * Authors: Jyrki Katajainen, Andreas Milton Maniotis, Bo Simonsen
 */
#endif __CPHSTL_RANK_ITERATOR__
#define __CPHSTL_RANK_ITERATOR__

#include <cassert>
#include <iterator>  // std::random_access_iterator_tag
#include <limits>    // std::numeric_limits
#include <type_traits>  // std::conditional, std::is const, std::remove_const

namespace cphstl {

    template <typename R>
    class rank_iterator {
    public:
        // associated types
        using owner_type = R;
        using opposite_type = typename std::conditional<std::is_const<R>::value, typename std::remove_const<R>::type, R const>::type;
```
using size_type = typename R::size_type;
using difference_type = typename R::difference_type;
using value_type = typename R::value_type;
using pointer = typename std::conditional<std::is_const<R>::value
    , value_type const*, value_type*>::type;
using reference = typename std::conditional<std::is_const<R>::value,
    typename R::const_reference, typename R::reference>::type;
using iterator_category = std::random_access_iterator_tag;

// friends
friend R;
friend class rank_iterator<opposite_type>;

private:

// classes

template <typename X, typename Y>
struct is_comparable;

template <typename X>
struct is_comparable<X, typename std::remove_const<X>::type> {
    class yes;
};

template <typename X>
struct is_comparable<X, typename std::add_const<X>::type> {
    class yes;
};

// constants

static size_type constexpr sentinel = std::numeric_limits<size_type>::max();

// variables

owner_type* owner_p;
size_type rank;

// parameterized constructor

rank_iterator(owner_type* p, size_type offset = sentinel) :
    owner_p(p), rank(offset) {
}

public:

// default constructor
rank_iterator()
 : owner_p(nullptr), rank(sentinel) {}

// copy constructors

// generated by the compiler if needed
// rank_iterator(const rank_iterator&) = default;
rank_iterator(rank_iterator<typename std::remove_const<R>::type> const& x)
 : owner_p(x.owner_p), rank(x.rank) {}

// assignments

// generated by the compiler if needed
// rank_iterator& operator=(rank_iterator const&) = default;
rank_iterator& operator=(rank_iterator<typename std::remove_const<R>::type> const& x) {
    owner_p = x.owner_p;
    rank = x.rank;
    return *this;
}

// destructor

~rank_iterator() {
};

// operator *

reference operator*() const {
    assert(owner_p != nullptr and rank != sentinel);
    return (*owner_p)[rank];
}

// operator ->

pointer operator->() const {
    assert(owner_p != nullptr and rank != sentinel);
    return &(*owner_p)[rank];
}

// operator ++; pre-increment

rank_iterator& operator++() {
    assert(owner_p != nullptr and rank != sentinel);
    ++rank;
    if (rank == (*owner_p).size()) {
        // Increment the iterator.
    }
}
    rank = sentinel;
    } return *this;
}

// operator ++: post-increment
rank_iterator operator++(int) {
    rank_iterator return_value(*this);
    operator++();
    return return_value;
}

// operator --: pre-decrement
rank_iterator& operator--() {
    assert(owner_p != nullptr and (*owner_p).size() != 0);
    if (rank == sentinel) {
        rank = (*owner_p).size() - 1;
    } else {
        --rank;
    } return *this;
}

// operator --: post-decrement
rank_iterator operator--(int) {
    rank_iterator return_value(*this);
    operator--();
    return return_value;
}

// operator +=
rank_iterator& operator+=(difference_type n) {
    assert(owner_p != nullptr);
    difference_type new_point = rank;
    if (rank == sentinel) {
        new_point = (*owner_p).size();
    }
    new_point += n;
    if (new_point < 0) {
        rank = sentinel;
        return *this;
    }
    rank = size_type(new_point);
    if (rank >= (*owner_p).size()) {
        rank = sentinel;
    }
    return *this;
}
// operator -=
rank_iterator& operator-=(difference_type n) {
    return operator+=(~n);
}

// operator +
rank_iterator operator+(difference_type n) const {
    rank_iterator temporary = *this;
    temporary.operator+=(n);
    return temporary;
}

// operator -
rank_iterator operator-(difference_type n) const {
    return operator+(-n);
}

// iterator distance

template<
type S, typename = typename is_comparable<S, R>::
    yes>
difference_type operator-(rank_iterator<S> const& other) const {
    assert(owner_p == other.owner_p);
    size_type y = rank;
    if (rank == sentinel) {
        y = (*owner_p).size();
    }
    size_type x = other.rank;
    if (other.rank == sentinel) {
        x = (*owner_p).size();
    }
    return y - x;
}

// operator ==

template<
type S, typename = typename is_comparable<S, R>::
    yes>
bool operator==(rank_iterator<S> const& other) const {
    return rank == other.rank and owner_p == other.owner_p;
}

// operator !=

template<
type S>
bool operator!=(rank_iterator<S> const& other) const {
    return not (*this == other);
template <typename S>
bool operator<(rank_iterator<S> const& other) const {
    return ((*this) - other) < difference_type(0);
}

// operator >

template <typename S>
bool operator>(rank_iterator<S> const& other) const {
    return other < *this;
}

// operator ≤

template <typename S>
bool operator<=(rank_iterator<S> const& other) const {
    return not (other < *this);
}

// operator ≥

template <typename S>
bool operator>=(rank_iterator<S> const& other) const {
    return not (*this < other);
}

// operator +(int, iterator)

template <typename R>
rank_iterator<R> operator+(typename R::difference_type n, rank_iterator<R> const& a) {
    return a + n;
}

#endif

// A predefined compare functor is provided for each type, for which
// operator< is defined.

Authors: Jyrki Katajainen, Michael Neidhardt, Bo Simonsen © 2009, 2016

*/
# ifndef __LEDA_COMPARE_FUNCTIONS__
# define __LEDA_COMPARE_FUNCTIONS__

#include <functional> // std::binary_function
#include <type_traits> // std::is_convertible
#include <utility> // std::pair

namespace cphleda {

    template <typename K, typename L = K>
    class comparator :
        public std::binary_function<K, L, int> {
        public:

            int operator() (K const& x, L const& y) const {
                static_assert(std::is_convertible<L, K>::value, "Only convertible types can be compared");
                if (x < y) {
                    return -1;
                }
                if (y < x) {
                    return 1;
                }
                return 0;
            }
    }

    template <typename T>
    class stl_compare_less :
        public std::binary_function<T, T, bool> {
        public:

            stl_compare_less(int (*_f)(T const&, T const&)) {
                (*this).f = _f;
            }

            bool operator() (T const& a, T const& b) const {
                if (((*this).f(a, b) == -1)) {
                    return true;
                }
                return false;
            }

        private:

            int (*f)(T const&, T const&);
        }

    template <typename P, typename I>
    class stl_compare_less_pair :
        public std::binary_function<P, I, bool> {
        };
}

public:
    stl_compare_less_pair(int (*_f)(P const&, P const&)) {
        (*this).f = _f;
    }
    bool operator() (std::pair<P, P> const & a, std::pair<P, P> const & b) const {
        if (((*this).f(a.first, b.first) == 1) {
            return true;
        } else return false;
    }
private:
    int (*_f)(P const&, P const&);
};

template<typename T>
int compare(T const& a, T const& b) {
    if (a < b) {
        return -1;
    } else if (a > b) {
        return 1;
    } return 0;
}

#ifdef __CPHSTL_COUNTING_ALLOCATOR__

#endif
counting-allocator.h++

/* -=- C++ -=-*/

An allocator that counts the number of bytes allocated and
delagates the actual work to std::allocator.

Define VERBOSE to get information on every allocation and
deallocation.

Authors: Jyrki Katajainen, Bjarke Buur Mortensen © 2001, 2012

*/

#include <iostream> // std streams
#include <memory> // std::allocator
#include <utility> // std::forward
// Use base class to have a single object counter

class counting_allocator_base {
public:

using count_type = unsigned long;

static count_type bytes_in_use() {
    return current_bytes;
}

static count_type allocators_in_use() {
    return allocators;
}

static count_type max_bytes_allocated() {
    return max_bytes;
}

// reset total statistics

static void reset_counts() {
    allocators = 0;
    current_bytes = 0;
    max_bytes = 0;
}

protected:

static count_type allocators; // number of allocators derived from this class
static count_type current_bytes; // current number of allocated bytes
static count_type max_bytes; // max number of bytes allocated
}

};

template <typename T>
class counting_allocator : public counting_allocator_base {
private:

count_type alloc_id; // id of the allocator
std::allocator<T> alloc;

count_type elements; // current number of elements allocated

count_type max_elements; // max number of elements allocated

public:

using value_type = typename std::allocator<T>::value_type;
using size_type = typename std::allocator<T>::size_type;
using difference_type = typename std::allocator<T>::difference_type;
using pointer = typename std::allocator<T>::pointer;
using const_pointer = typename std::allocator<T>::const_pointer;
using reference = typename std::allocator<T>::reference;
using const_reference = typename std::allocator<T>::const_reference;

pointer address(reference r) const {
    return &r;
}

counting_allocator() throw() :
    alloc(), elements(0), max_elements(0) {
    alloc_id = allocators++;
}

template<typename U>
counting_allocator(counting_allocator<U> const&) throw() :
    alloc(), elements(0), max_elements(0) {
    alloc_id = allocators++;
}

// note that, if allocation fails, we update our variables wrongly

pointer allocate(size_type n) {
    elements += n;
    current_bytes += n * sizeof(T);
    if (max_elements < elements) {
        max_elements = elements;
    }
    if (max_bytes < current_bytes) {
        max_bytes = current_bytes;
    }
}

#endif VERBOS

std::cout << __PRETTY_FUNCTION__ << std::endl;
std::cout << ": alloc(), elements(0), max_elements(0) { 
    alloc_id = allocators++;
}

template <typename U>
counting_allocator(counting_allocator<U> const&) throw() :
    alloc(), elements(0), max_elements(0) {
    alloc_id = allocators++;
}

// note that, if allocation fails, we update our variables wrongly

pointer address(reference r) const {
    return &r;
}

counting_allocator() throw() :
    alloc(), elements(0), max_elements(0) {
    alloc_id = allocators++;
}

template<typename U>
counting_allocator(counting_allocator<U> const&) throw() :
    alloc(), elements(0), max_elements(0) {
    alloc_id = allocators++;
}

// note that, if allocation fails, we update our variables wrongly

pointer allocate(size_type n) {
    elements += n;
    current_bytes += n * sizeof(T);
    if (max_elements < elements) {
        max_elements = elements;
    }
    if (max_bytes < current_bytes) {
        max_bytes = current_bytes;
    }
}

#endif VERBOS
return alloc.allocate(n);
}

void deallocate(pointer p, size_type n) {
    alloc.deallocate(p, n);
    elements -= n;
    current_bytes -= n * sizeof(T);
}

#ifdef VERBOSE
    std::cout << __PRETTY_FUNCTION__ << std::endl;
    std::cout << "\" << alloc_id << \"\" << std::endl
    << "Deallocating " << n
    << " elements (" << n * sizeof(T) << \" \" bytes)" << std::endl
    << \"\t\t\" << elements << \" \" (n
    << elements * sizeof(T) << \" \" bytes)" << std::endl;
    std::cout << \"\tTotal = \" << elements << \" \" bytes" << std::endl;

#endif

}

void construct(pointer p, const T& val) {
    alloc.construct(p, val);
}

void construct(pointer p, T&& val) {
    alloc.construct(p, std::forward<T>(val));
}

void destroy(pointer p) {
    alloc.destroy(p);
}

size_type max_size() const throw() {
    return alloc.max_size();
}

template<typename U>
struct rebind {
    using other = counting_allocator<U>;
};

// accounting functions

count_type elements_allocated() {
    return elements;
}

count_type max_elements_allocated() {
    return max_elements;
}
count_type bytes_allocated() {
    return elements_allocated() * sizeof(T);
}

count_type max_bytes_allocated() {
    return max_elements_allocated() * sizeof(T);
};

// initialize static members

counting_allocator_base::counting_allocator_base::count_type counting_allocator_base::allocators = 0;
counting_allocator_base::counting_allocator_base::count_type counting_allocator_base::current_bytes = 0;
counting_allocator_base::counting_allocator_base::count_type counting_allocator_base::max_bytes = 0;

// swap_based.i++

#include <utility>  // std::swap

namespace swap_based {

    template <typename iterator>
    void __reverse(iterator first, iterator last) {
        while (true) {
            if (first == last or first == --last) {
                return;
            } else {
                std::swap(*first, *last);
                ++first;
            }
        }
    }

    template <typename S>
    void reverse(S& s) {
        __reverse(s.begin(), s.end());
    }

// move_based.i++

#include <utility>  // std::move
namespace move_based {

    template <typename S, typename T>
    void reverse_copy(S& input, T& output) {
        auto n = input.size();
        while (n != 0) {
            --n;
            output.push_back(std::move(input[n]));
            input.pop_back();
        }
    }

    template <typename S>
    void reverse(S& s) {
        S tmp;
        reverse_copy(s, tmp);
        s.swap(tmp);
    }
}

#include <vector>
using X = std::vector<V, A>;

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

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

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

#include "space_efficient_array.h++"
using X = cphstl::space_efficient_array<V, A>;
Drivers

reverse-driver.c++

```cpp
#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

#ifdef MEASURE_MOVES
long volatile moves = 0;
#endif

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;
  }
```

template<typename T> bool operator<(move_counter<T> const &x, move_counter<T> const &y) {
    return x.datum < y.datum;
}

template<typename T>
bool operator<=(move_counter<T> const &x, move_counter<T> const &y) {
    return x.datum <= y.datum;
}

#endif

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

int main(int argc, char **argv) {
    #ifdef MEASURE_MOVES
        using V = move_counter<int>;
    #else
        using V = int;
    #endif
using A = std::allocator<V>;

#ifdef STD

using X = std::vector<V, A>;
#else
using X = cphstl::STRUCTURE<V, A>;
#endif

unsigned int n = 0;
if (argc == 2) {
    n = atoi(argv[1]);
} else {
    usage(argc, argv);
}
if (n < 1 || n > MAXSIZE) {
    std::cerr << "n out of bounds [1.." << MAXSIZE << "]\n";
    usage(argc, argv);
};
unsigned int const repetitions = MAXSIZE / n;
X* many = new X[repetitions];
for (volatile unsigned int t = 0; t != repetitions; ++t) {
    for (unsigned int i = 0; i != n; ++i) {
        many[t].push_back(i);
    }
}
#ifdef MEASURE_MOVES

moves = 0;
#else

std::clock_t start = std::clock();
#endif

for (volatile unsigned int t = 0; t != repetitions; ++t) {
    ALGORITHM::reverse(many[t]);
}
#endif

#ifdef MEASURE_MOVES

std::clock_t stop = std::clock();
#endif
#endif
double scale = double(repetitions) * double(n);

#if defined(MEASURE_MOVES)
    std::cout.precision(3);
    std::cout << n << 't' << double(moves) / double(scale) << std::endl;
#else
    double ns = 1.0e9 * double(stop - start) / double(CLOCKS_PER_SEC);
    std::cout << n << 't' << ns / double(scale) << std::endl;
#endif

delete[] many;
return 0;
}

space-driver.cpp

// Measures the amount of space taken up by a data structure
// Authors: Jyrki Katajainen, Bjarke Buur Mortensen © 2001, 2012

using V = int;
using A = counting_allocator<V>;

float run(std::size_t n) {
    assert(RAND_MAX > n);
    counting_allocator_base::reset_counts();
    X container;
    for (std::size_t i = 0; i != n; ++i) {
        container.push_back(V(random()));
    }
    assert(container.size() == n);
    return float(counting_allocator_base::bytes_in_use());
}

int main(int argc, char* argv[]) {

srnd(1837362);
for (std::size_t n = 100000; n <= 10000000; n += 100000) {
    std::cout.setf(std::ios::fixed, std::ios::floatfield);
    std::cout.precision(3);
    float bytes_in_use = run(n);
    float overhead = bytes_in_use - float(n * sizeof(V));
    float in_procents = 100.0 * overhead / float(n * sizeof(V));
    std::cout << n << "\t" << in_procents << std::endl;
}
return 0;

sort-driver.c++

#include <iostream>
#include <algorithm>
#include <cmath>
#include <ctime>
#include <functional>
#include <iterator>
#include <memory>
#include <vector>

extern int ilogb(double) throw();

template<typename iterator>
bool is_permutation(iterator first, iterator beyond) {
    using V = typename std::iterator_traits<iterator>::value_type;
    std::vector<V> copy(first, beyond);
    std::sort(copy.begin(), copy.end());
    for (auto q = copy.begin(); q != copy.end(); ++q) {
        V i = V(q - copy.begin());
        if (*q != i) {
            std::cerr << i << " element missing " << *q << " instead\n";
            std::cerr << "n: " << beyond - first << std::endl;
            return false;
        }
    }
    return true;
}

template<typename iterator, typename comparator>
bool is_sorted(iterator a, iterator o, comparator less) {
    using Z = typename std::iterator_traits<iterator>::difference_type;
    Z const n = o - a;
```cpp
bool violated = false;
if (n < 2) {
    return true;
}
for (Z i = n - 1; i > 0; --i) {
    if (less(*(a + i), *(a + i - 1))) {
        std::cerr << i << ": me " << *(a + i) << " before " << *(a + i - 1) << std::endl;
        violated = true;
    }
}
return not violated;
}

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) {
    using C = std::less<V>;
    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);
    }
    X b;
    b.reserve(MAXSIZE);
    unsigned int const repetitions = MAXSIZE / n;
    for (unsigned int i = 0; i ≠ repetitions; ++i) {
        for (unsigned int i = 0; i ≠ n; ++i) {
            b.push_back(V(i));
        }
        std::random_shuffle(b.end() - n, b.end());
    }
    auto c = b.begin();
    std::clock_t start = std::clock();
    for (volatile unsigned int t = 0; t ≠ repetitions; ++t) {
```
# if defined(INTROSORT)
std::sort(c, c + n, C());
# endif

# if defined.HEAPSORT)
std::partial_sort(c, c + n, c + n, C());
# endif

c = c + n;
}
std::clock_t stop = std::clock();

auto d = b.begin();
for (volatile unsigned int t = 0; t != repetitions; ++t) {
  bool ok = ::is_sorted(d, d + n, C());
  if (! ok) {
    return 1;
  }
  ok = ::is_permutation(d, d + n);
  if (! ok) {
    return 2;
  }
  d = d + n;
}

double scale = double(repetitions) * double(n) * double(ilogb(n));
double ns = 1.0e9 * double(stop - start) / double(CLOCKS_PER_SEC);
std::cout.precision(3);
std::cout << n << '\t' << ns / double(scale) << '\n';
return 0;
}

**scan-driver.c++**

# if ! defined(MAXSIZE)
#define MAXSIZE (128 * 1024 * 1024)
# endif

#include <cassert> // assert macro
#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] << " " << argv[1];
    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 << "]" << std::endl;
        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 (volatile unsigned int t = 0; t != repetitions; ++t) {
        auto c = v[t].begin();
        auto e = v[t].end();
        for (; c != e; ++c) {
            *c = V(0);
        }
    }
    std::clock_t stop = std::clock();
    for (unsigned int t = 0; t != repetitions; ++t) {
        for (unsigned int i = n; i != 0; ) {
            --i;
            assert(v[t][i] == V(0));
            v[t].pop_back();
        }
    }
    double ns = 1.0e9 * double(stop - start) / double(CLOCKS_PER_SEC);
```cpp
std::cout.precision(3);
std::cout << n << '\t' << ns / double(repetitions * n) << '\n';
return 0;
}

#include <cassert> // assert macro
#include <ctime>   // std::clock_t, std::clock, CLOCKS_PER_SEC
#include <iostream> // std::cout, std::cerr
#include <algorithm> // 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";
    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 || 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(i);
        }
    }
```
unsigned int const prime = 617;
unsigned int const mask = n - 1;
std::clock_t start = std::clock();
for (volatile unsigned int t = 0; t ≠ repetitions; ++t) {
    auto c = v[t].begin();
    if (c ≠ v[t].end()) {
        *(c + i) = V(0);
    }
}
for (unsigned int i = prime; i ≠ 0; i = (i + prime) bitand mask)
    *(c + i) = V(0);
std::clock_t stop = std::clock();
for (unsigned int t = 0; t ≠ repetitions; ++t) {
    for (unsigned int i = n; i ≠ 0; ) {
        --i;
        assert(v[t][i] == V(0));
        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;

---

grow-driver.c++

#define MAXSIZE (128 * 1024 * 1024)

#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);
```c
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++

```c
#include <ctime> // std::clock_t, std::clock, CLOCKS_PER_SEC
#include <iostream> // std::cout, std::cerr
```
```cpp
#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 || 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 = n; i != 0; --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 << n << "'t' << ns / double(repetitions * n) << 'n';
    return 0;
}
```
/*
   Advice: A good guess is a must!
*/

#include <iostream> // standard streams
#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] << " [<guess>]\n";
    exit(1);
}

int main(int argc, char** argv) {
    unsigned int mega = 1024 * 1024;
    std::vector<X> v;
    unsigned int t = 0;
    if (argc == 1) {
    }
    else if (argc == 2) {
        t = atoi(argv[1]);
    }
    else {
        usage(argc, argv);
    }
    while (true) {
        unsigned int n = 0;
        try {
            for (; n != t; ++n) {
                v.push_back(std::move(X()));
                v[n].push_back(V(n));
            }
        }
        catch (...) {
            for (unsigned int i = 0; i != n; ++i) {
                v[i].clear();
            }
            v.clear();
            t = t - mega;
            continue;
        }
        break;
    }
}
while (true) {
    try {
        std::cout << t << std::endl;
        v.push_back(std::move(X()));
        for (unsigned int n = 0; n != mega; ++n) {
            v[t].push_back(V(n));
        }
        for (unsigned int n = 0; n != mega - 1; ++n) { // leave one!
            v[t].pop_back();
        }
        ++t;
    } catch (...) {
        std::cerr << t << std::endl;
        break;
    }
}
return 0;

#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) {
    if (argc != 2) {
        usage(argc, argv);
    }
    unsigned int n = atoi(argv[1]);
    if (n < 1 or n > MAXSIZE) {
        std::cerr << "n out of bounds [1.." << MAXSIZE << "]\n";
        exit(2);
    }
std::vector<X> v;
unsigned int const repetitions = MAXSIZE / n;
v.resize(repetitions);
std::vector<unsigned int> gap;
for (unsigned int t = 0; t != repetitions; ++t) {
    V* previous = nullptr;
    for (unsigned int i = 0; i != n; ++i) {
        v[t].push_back(V(i));
        V* current = &v[t].back();
        if ((current - previous) > 4 || (previous - current) > 4) {
            gap.push_back(i);
        }
        previous = current;
    }
    unsigned int gaps_per_case = gap.size() / repetitions;
    std::cout << "n, gaps: " << n << "", " << gaps_per_case << "\n";
    unsigned int const rounds = 20 * n / gaps_per_case;
    auto p = gap.begin();
    std::clock_t start = std::clock();
    for (unsigned int t = 0; t != repetitions; ++t) {
        for (unsigned int i = 0; i != n; ++i) {
            v[t].push_back(V(i));
            if (i == *p) {
                for (volatile unsigned int j = 0; j != rounds; ++j) {
                    v[t].pop_back();
                    v[t].push_back(V(i));
                }
                ++p;
            }
        }
        while (v[t].size() > 0) {
            v[t].pop_back();
        }
    }
    std::clock_t stop = std::clock();
    double scale = double(21 * n * repetitions);
    double ns = 1.0e9 * double(stop - start) / double(CLOCKS_PER_SEC);
    std::cout.precision(3);
    std::cout << n << '\t' << ns / scale << '\n';
}
return 0;
Makefile

CXX=g++
CXXFLAGS=-O3 -Wall -DNDEBUG -std=+-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))

N = 1024 32768 1048576 33554432

# Variants for reversals: -DMEASURE_MOVES -DSTD

$(swap): %.swap-reverse : %.i++
$CXX $(CXXFLAGS) -DALGORITHM=swap_based -DSTRUCTURE=$*

@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 $(CXXFLAGS) -DALGORITHM=move_based -DSTRUCTURE=$*

@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++
99

```bash
$\$(CXX) \$(CXXFLAGS) -DINTROSORT sort-driver.c++
@for n in \$(N) ; do \\
    ./a.out $$n ; \\
done; \\
rm -f ./a.out data-structure.i++

$\$(heapsort): %.heapsort : %.i++
@cp $*.*.i++ data-structure.i++
$\$(CXX) $\$(CXXFLAGS) -DHEAPSORT sort-driver.c++
@for n in \$(N) ; do \\
    ./a.out $$n ; \\
done; \\
rm -f ./a.out data-structure.i++

$\$(scan): %.scan : %.i++
@cp $*.*.i++ data-structure.i++
$\$(CXX) $\$(CXXFLAGS) scan-driver.c++
@for n in \$(N) ; do \\
    ./a.out $$n ; \\
done; \\
rm -f ./a.out data-structure.i++

$\$(jump): %.jump : %.i++
@cp $*.*.i++ data-structure.i++
$\$(CXX) $\$(CXXFLAGS) jump-driver.c++
@for n in \$(N) ; do \\
    ./a.out $$n ; \\
done; \\
rm -f ./a.out data-structure.i++

$\$(grow): %.grow : %.i++
@cp $*.*.i++ data-structure.i++
$\$(CXX) $\$(CXXFLAGS) grow-driver.c++
@for n in \$(N) ; do \\
    ./a.out $$n ; \\
done; \\
rm -f ./a.out data-structure.i++

$\$(shrink): %.shrink : %.i++
@cp $*.*.i++ data-structure.i++
$\$(CXX) $\$(CXXFLAGS) shrink-driver.c++
@for n in \$(N) ; do \\
    ./a.out $$n ; \\
done; \\
rm -f ./a.out data-structure.i++

$\$(many): %.many : %.i++
@cp $*.*.i++ data-structure.i++
$\$(CXX) $\$(CXXFLAGS) many-driver.c++
./a.out
rm -f ./a.out data-structure.i++
```
$(\text{gap}): \%.gap : \%.i++
@cp $*.i++ data-structure.i++
$(\text{CXX})$ $(\text{CXXFLAGS})$ gap-driver.c++
@for n in $(\text{N})$ ; do \n   ./a.out $n ; \n done; \nrm -f ./a.out data-structure.i++
clean:
   - rm -f temp core a.out algorithm.i++ data-structure.i++ *o 2>/dev/null
   dev/null
veryclean: clean
   - rm -f */*~ 2>/dev/null
find:
   find . -type f -print -exec grep $(\text{word}) {} \; | less