Göteborg, 12 May 2004
Corrections, 16 May 2004

Title:

The cost of iterator validity

Speaker:

Jyrki Katajainen
University of Copenhagen

These slides are available at

`http://www.cphstl.dk/`.

Announcement

SWAT 2004

Invited speakers:

* Gerth S. Brodal, University of Aarhus
* Charles E. Leiserson, MIT

Website: http://swat.diku.dk/

OLA 2004

Invited speakers:

* Allan Borodin, University of Toronto
* Anna Karlin, University of Washington

Website: http://ola.imada.sdu.dk/

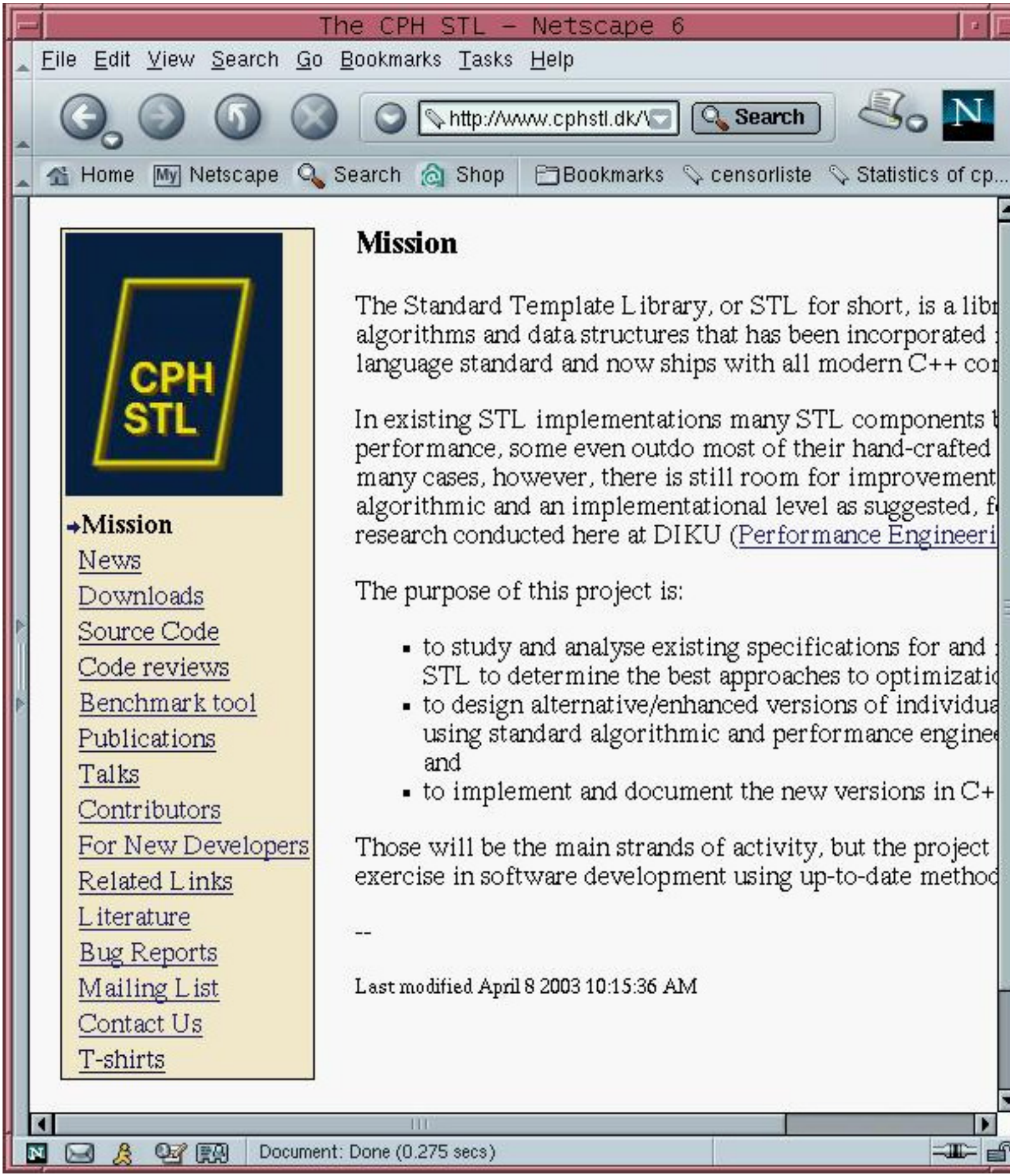
Summer School on Exp. Algorithmics

Invited speakers:

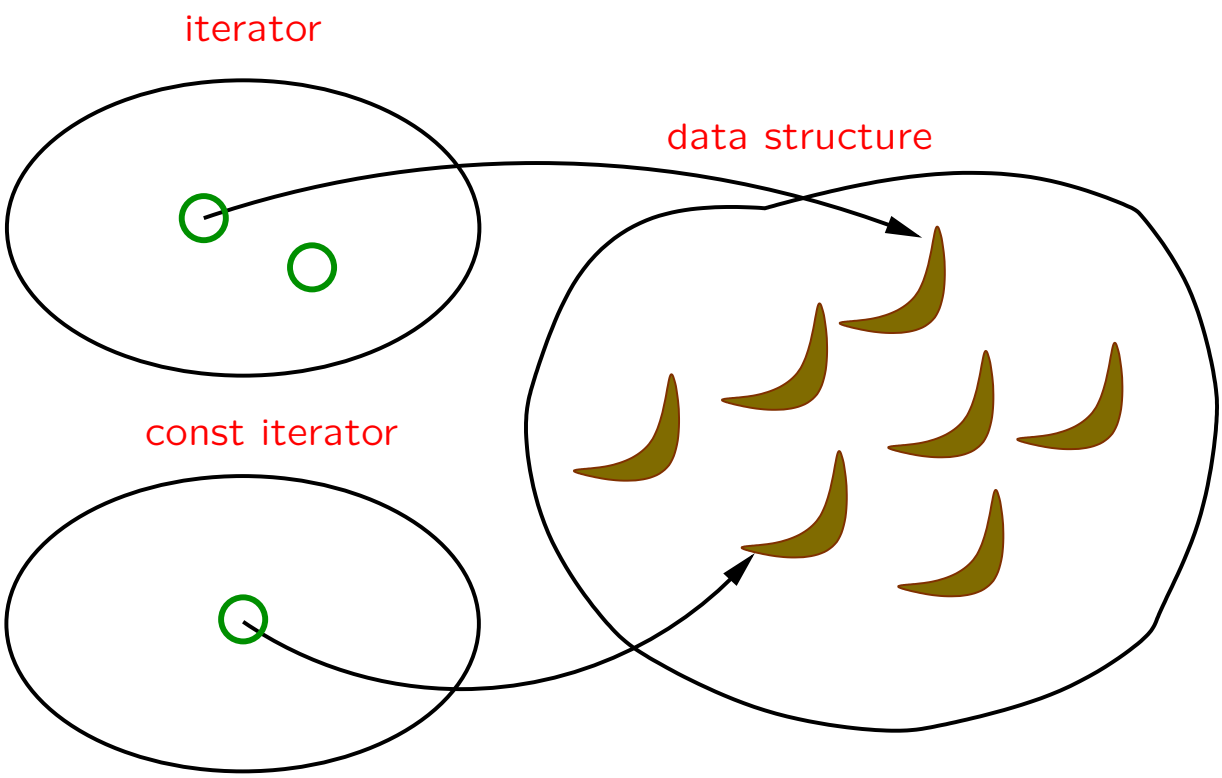
* Hervé Brönnimann, Polytechnic Univ.
* Peter Sanders, Max-Planck-Institut
* Alexander Stepanov, Adobe Systems Inc.

Website:

http://www.diku.dk/forskning/performance-engineering/Sommerskole/


The CPH STL - Netscape 6
File Edit View Search Go Bookmarks Tasks Help
http://www.cphstl.dk/
Search
Home Netscape Search Shop Bookmarks censorliste Statistics of cp...
CPH STL
Mission
News
Downloads
Source Code
Code reviews
Benchmark tool
Publications
Talks
Contributors
For New Developers
Related Links
Literature
Bug Reports
Mailing List
Contact Us
T-shirts
Mission
The Standard Template Library, or STL for short, is a libr
algorithms and data structures that has been incorporated
language standard and now ships with all modern C++ co
In existing STL implementations many STL components
performance, some even outdo most of their hand-crafted
many cases, however, there is still room for improvement
algorithmic and an implementational level as suggested, f
research conducted here at DIKU (Performance Engineeri
The purpose of this project is:
to study and analyse existing specifications for and
STL to determine the best approaches to optimizati
to design alternative/enhanced versions of individua
using standard algorithmic and performance engine
and
to implement and document the new versions in C+
Those will be the main strands of activity, but the project
exercise in software development using up-to-date metho
--
Last modified April 8 2003 10:15:36 AM
Document: Done (0.275 secs)

Common picture



Concept jungle

word used	reference
pointer	C language
address	assembly language
reference	C++ language
smart pointer	e.g. [Meyers 1996]
iterator	STL
item	LEDA
finger	algorithmic literature
position	[Aho et al. 1983]
handle	[Cormen et al. 2001]
locator	[Goodrich & Tamassia 1998]
tag	[Hagerup & Raman 2002]

Iterators

`X`: iterator type whose value type is `T`
`p, q`: objects of type `X`
`r`: object of type `X&`
`t`: object of type `T`

Category	Allowed expressions
trivial	`X p` (default constructor)
	`X()` (default constructor)
	`*p` (element load; read)
	`*p = t` (element store; write)
	`p->m` (equivalent to `(*p).m`)
forward	all earlier operations
	`X(p)` (copy constructor)
	`X p(q)` (copy constructor)
	`X p = q` (copy constructor)
	`p == q` (equality)
	`p != q` (inequality)
	`r = p` (assignment)
	`++r` (pre-increment)
	`r++` (post-increment)
	`*r++ (T t = *r; ++r; return t;)`

Iterators (cont.)

`i`: object of `X`'s difference type

Category	Allowed expressions
bidirectional	all earlier operations `--r` (pre-decrement) `r--` (post-decrement) `*r--` (`T t = *r; --r; return t;`)
random access	all earlier operations `p < q` (less) `p > q` (greater) `p <= q` (less or equal) `p >= q` (greater or equal) `r += i` (iterator addition) `p + i` (iterator addition) `i + p` (iterator addition) `r -= i` (iterator subtraction) `p - i` (iterator subtraction) `q - p` (difference) `p[i]` (equivalent to `*(p + i)`)

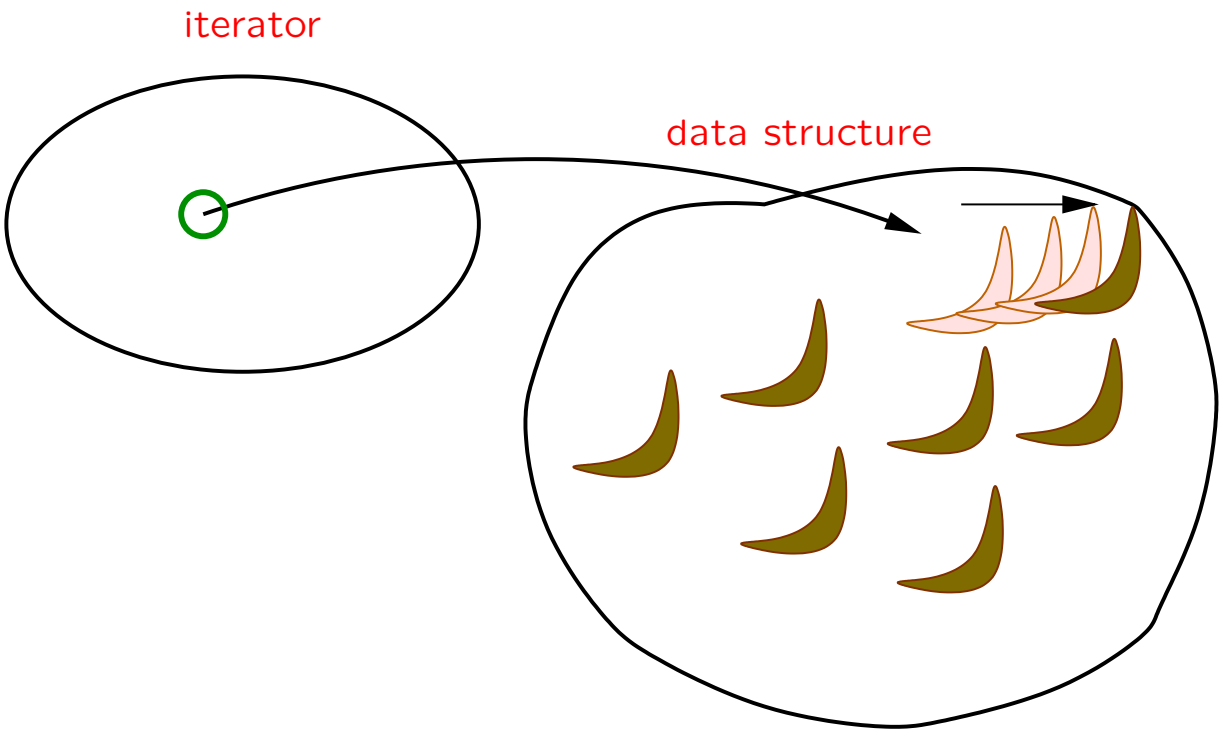
Relevance

- This "algebra" of iterators is fundamental to practically everything else in the Standard Template Library (STL). [Plauger et al. 2001, p. 26]
- Am implicit requirement for all iterators is that operations on them have no surprising overheads. [Plauger et al. 2001, p. 23]

On-line exercise: What is constant?

```
shell> cat exercise.c++
int main () {
  int* const p = 0;
  const int* q = p;
  const int* const r = p;
  int const* s = q;
}
shell> g++-3 exercise.c++
shell>
```

Iterator validity



Definition: An iterator and the element pointed to live in a close symbiosis; when the element is moved, the iterator may become invalid if it is not updated accordingly. A data structure is said to provide **iterator validity** if the iterators to its elements are kept valid at all times independent of the element moves.

Target data structures

abstract data structure	concrete data structure	STL name
ranked sequence	dynamic array	`vector, deque`
positional sequence	linked list	`list`
unordered dictionary	hash table	`hash_[multi]{set|map}`
ordered dictionary	balanced search tree	`[multi]{set|map}`
priority queue	heap	`priority_queue`

Element ordering: rank, position, comparator, insertion, arbitrary

Iterator strength: trivial, forward, bidirectional, random access

How would you provide iterator validity?

One possible solution

Restrict the use of iterators:

Aho et al. 1983: `print()` is an atomic operation.

LEDA rule: An iteration over the items in a collection C must not add new items to C. It may delete the item under the iterator, but no other item. The attributes of the items in C can be changed without restriction.

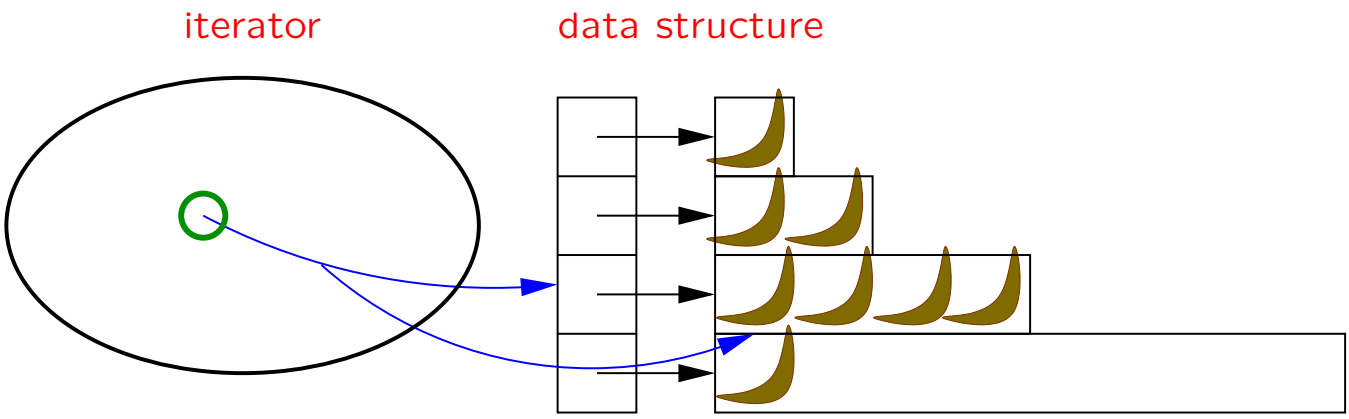
Available in the SGI STL

data structure	iterator strength	validity
`vector, deque`	random access	no
`list`	bidirectional	yes*
`hash_[multi]set`	const forward	no
`hash_[multi]map`	forward, not mutable	no
`[multi]set`	const bidirectional	yes*,**
`[multi]map`	bidirectional, not mutable	yes*,**
`priority_queue`	no iterators	no

* Deletions invalidate only the iterators to the erased elements.

** Iterator operations take constant **amortized** time for a sequence of ++ operations, but not for a sequence of ++ and -- operations.

Vector



Use the levelwise-allocated piles by Katajainen and Mortensen [2001]:

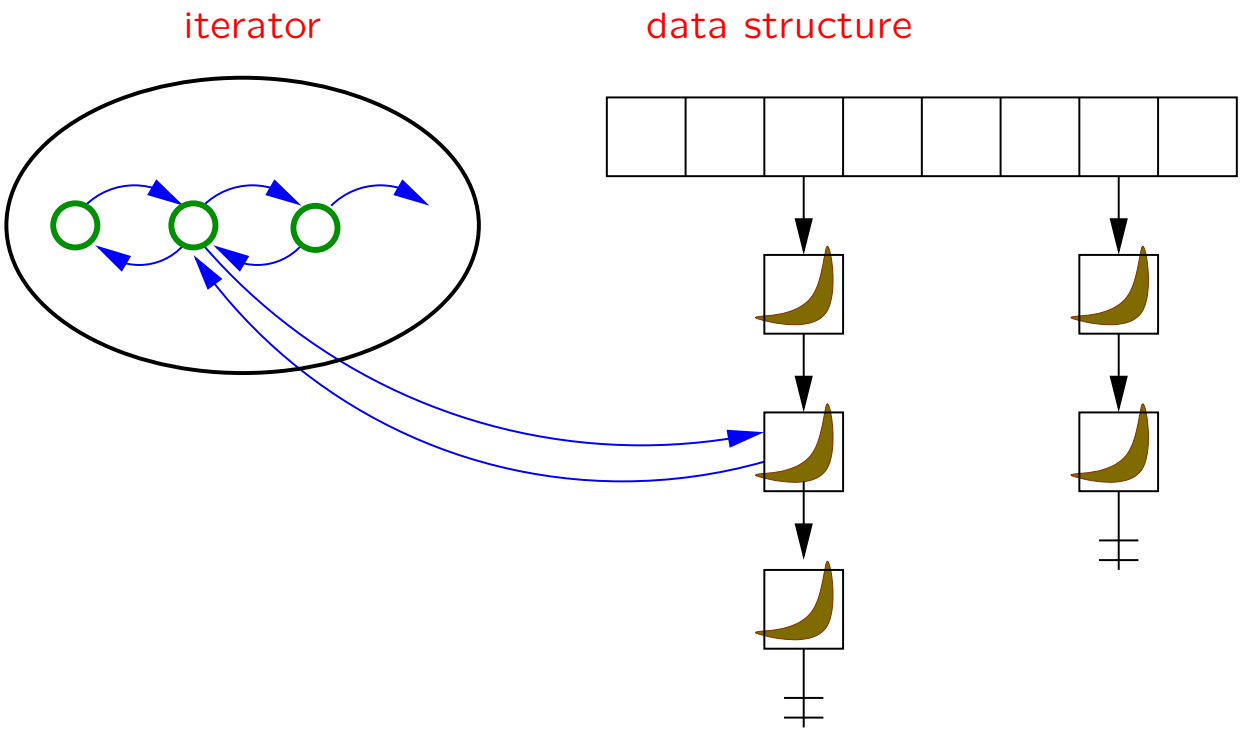
- `push_back()` and `pop_back()` require $O(1)$ worst-case time.
- Elements need not be moved due to the dynamization.
- `insert()` and `erase()` take $O(\sqrt{n})$ worst-case time.
- Represent an iterator as a level, position pair. This way all random-access-iterator operations take $O(1)$ worst-case time.
- `insert()` and `erase()` invalidate all iterators; `push_back()` and `pop_back()` keep the iterators valid.

Deque

Use three levelwise-allocated piles as proposed by Katajainen and Mortensen [2001]:

- `push_back()` and `pop_back()` require $O(1)$ worst-case time.
- `pop_back()` moves at most $O(1)$ elements, but these moves do not change the iterator ordering.
- `insert()` and `erase()` take $O(\sqrt{n})$ worst-case time.
- As for vector, represent an iterator as a level, position pair to support random-access-iterator operations in $O(1)$ worst-case time. The two half-full blocks in the middle need special handling.
- `insert()` and `erase()` invalidate all iterators, `push_back()` keeps the iterators valid, and `pop_back()` updates the iterators for the elements moved.

Hash table



Rely on linear hashing. This guarantees that in connection with each `erase()` and `insert()` $O(1)$ element moves are done on an average.

- When an element is erased, its iterator is erased from the iterator list.
- When an element is inserted, its iterator is inserted into the iterator list too.
- When an element is moved in a bucket split or merge, its iterator is also moved. It is easy to determine where the moved elements should be placed.

Balanced search tree

There are at least two options:

1. Use a leaf-oriented search tree when implementing `[multi]{set|map}`.
2. Use the iterator list technique as for hash tables.

Priority queue

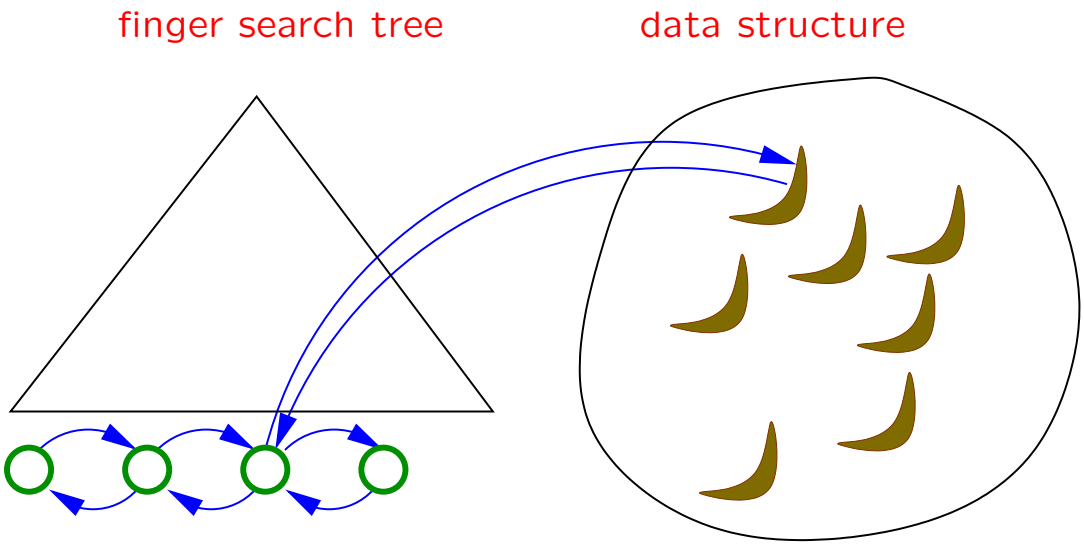
- Trivial iterators would make it possible to provide the operations `delete(p)` and `increase_priority(p)` that are missing in the specification given in the C++ standard.
- Bidirectional iterators could be provided with the iterator list technique. Normally, in heap operations element swaps are performed. These are easy to handle since each element knows the position of its iterator in the iterator list, and vice versa.
- Note that elements are iterated in arbitrary order. The maintenance of the elements in sorted order would be more expensive.

Elegance in the CPH STL

data structure	iterator strength
`resizable_array`	random access
`doubly_resizable_array`	random access
`list`	bidirectional
`hash_[multi]set`	const bidirectional
`hash_[multi]map`	bidirectional, not mutable
`[multi]set`	const bidirectional
`[multi]map`	bidirectional, not mutable
`priority_queue`	bidirectional

- Data structures provide iterator validity.
- All iterator operations take $O(1)$ worst-case time.
- Data structures require linear space, linear on the number of elements stored.
- None of the iterator operations make the data structure operations asymptotically more expensive.

Iterator-valid vector: alternative 1



- Give a tag for each element (related to its rank) and keep the tags in a finger search tree. An iterator is a leaf in this tree. Use the tags for iterator comparisons.
- Adapt the tag universe (size n^3) with the number of elements stored (n) by performing rebuildings in background.
- Utilize a finger search when performing the iterator additions `p + i` etc.
- The cost of all iterator operations is $O(1)$ in the worst case, except that of iterator addition which takes $O(\log \mathtt{i})$ time.

Problem: I do not know any implementation of the finger search trees by Brodal et al. [2003] or Dietz and Raman [1994].

Iterator-valid vector: alternative 2

Instead of finger search trees use search trees guaranteeing $O(1)$ update time. This would increase the time needed for iterator additions to $O(\log n)$, keeping the cost of other iterator operations unchanged.

Problem: I have not seen any implementation of search trees by Levcopoulos and Overmars [1988] or Fleischer [1996].

Iterator-valid vector: alternative 3

Instead of finger search trees use normal balanced search trees. This is implementable, but it increases the cost of iterator operations to $O(\log n)$.

Iterator-valid vector: lower bound

`X`: iterator type whose value type is `T`
`p, q`: objects of type `X`
`t`: object of type `T`
`V`: vector storing objects of type `T`

At least one of the modification operations `insert(p,t)`, `erase(q)`, and iterator operation `p - V.begin()` has to take $\Omega(\log n / \log \log n)$ amortized time. The proof is by reduction to the subset rank problem.

Conclusions

- In the C++ standard the general modification operations specified for vector seem to be too strong to get iterator validity and $O(1)$-time iterator operations at the same time.
- Is it possible to devise a vector with $O(1)$-time iterator operations and $O(\log n)$-time modification operations at the same time?
- Be aware that we have assumed that memory allocation and memory deallocation functions can be executed in $O(1)$ worst-case time.
- It is interesting to point out that bidirectional iterators with `operator<` have been studied earlier. All operations for such iterators can be realized in worst-case $O(1)$ time [Dietz & Sleator 1987].
- One may want to get a snapshot of the corresponding container at the time the iteration is started. This would require persistent data structures. Is this type of validity relevant in practice?