Göteborg, 11 June 2001
Corrections, 15 June 2001

Title:
Space-efficient vectors and deques

Speaker:
Jyrki Katajainen
Datalogisk Institut
Københavns Universitet

Co-worker:
Bjarke Buur Mortensen

5th Workshop on Algorithm Engineering
(to appear)
Background: the Copenhagen STL

Project start: September 2000

Goal: alternative/enhanced versions of individual STL components

Contributors: ca. 20 students have written parts of the library

Status: first implementations for the most interesting modules exist

Emphasis: performance engineering, software engineering, algorithmics

Availability: http://cphstl.dk

Current problem: How to transfer the existing prototypes to a product?
std::vector in the C++ library

Required by the C++ standard

- sequence operations in $O(1)$ amortized time
- modifying operations in linear time
- according to a technical correction elements must be stored contiguously

SGI STL implementation

- standard doubling technique
- unbounded extra space
- $n$ push_backs require $\Theta(n)$ element moves
**std::deque in the C++ library**

- `front()` operator[]( )
- `back()`
- `push_front()`
- `push_back()`
- `pop_front()`
- `pop_back()`
- `insert()`
- `erase()`

**Required by the C++ standard**

- sequence operations in $O(1)$ worst-case time
- modifying operations in $O(\min\{i, n-i\})$ time, where $i$ is the insertion/erasure point

**SGI STL implementation**

- two levels: index blocks and data blocks; data blocks are of a fixed size; only the two extreme data blocks can be non-full
- unbounded extra space
- $O(1)$ amortized time push operations
Earlier results

Vectors and deques [Brodnik et al., 1999]

- sequence operations in $O(1)$ worst-case time
- $O(\sqrt{n})$ extra space (measured in elements and in objects of the built-in types)
- $\Omega(\sqrt{n})$ is a lower bound for the amount of extra space needed

Vectors [Goodrich and Kloss II, 1999]

- modifying operations in $O(n^\varepsilon)$ amortized time for any fixed constant $\varepsilon > 0$

Deques [Mortensen, 2001]

- some implementation details were missing in [Brodnik et al., 1999]
- after filling in these details the implementation got complicated
Piles (and heaps)

Shape property:

Representation property:

Capacity property:

Order property:

© Performance Engineering Laboratory
Levelwise-allocated piles

- sequence operations in $O(1)$ worst-case time
- element with index $k \in [0..n-1]$ has index $k - 2^\lceil \log_2(k+1) \rceil + 1$ at level $\lfloor \log_2(k+1) \rfloor$
- $O(n)$ extra space
- elements are never moved by push_back or pop_back
Blockwise-allocated piles

- sequence operations in $O(1)$ worst-case time
- $O(\sqrt{n})$ extra space
- elements are never moved by push_back or pop_back
Faster modifying operations

[Goodrich and Kloss II, 1999]

insert element $s$ between $r$ and $t$

- modifying operations in $O(\sqrt{n})$ worst-case time
- in the twin-pile we have to store double as many pointers
Space-efficient deques

Everything is easy until $A$ or $B$ gets empty.
What if \( A \) gets empty?

**Observation:** A space-efficient vector can be constructed backwards, this can be done piecewise, and the structure can be used simultaneously during such a construction.

![Diagram](image)

- sequence operations in \( O(1) \) worst-case time
- modifying operations in \( O(\sqrt{n}) \) time
- \( O(\sqrt{n}) \) extra space
Some experimental results

<table>
<thead>
<tr>
<th>container</th>
<th>push_back (ns)</th>
<th>pop_back (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>std::deque</td>
<td>85</td>
<td>11</td>
</tr>
<tr>
<td>std::vector</td>
<td>115</td>
<td>2</td>
</tr>
<tr>
<td>our deque</td>
<td>113</td>
<td>35</td>
</tr>
<tr>
<td>our deque (with reorganization)</td>
<td>113</td>
<td>375</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>container</th>
<th>sequential access (ns)</th>
<th>random access (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>std::deque</td>
<td>117</td>
<td>210</td>
</tr>
<tr>
<td>std::vector</td>
<td>2</td>
<td>60</td>
</tr>
<tr>
<td>our deque</td>
<td>56</td>
<td>160</td>
</tr>
<tr>
<td>our deque (with reorganization)</td>
<td>58</td>
<td>162</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>container</th>
<th>1000 inserts (s)</th>
<th>1000 inserts (s)</th>
<th>1000 inserts (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>initial size</td>
<td>initial size</td>
<td>initial size</td>
</tr>
<tr>
<td>std::deque</td>
<td>0.07</td>
<td>1.00</td>
<td>17.5</td>
</tr>
<tr>
<td>std::vector</td>
<td>0.015</td>
<td>0.61</td>
<td>12.9</td>
</tr>
<tr>
<td>our deque</td>
<td>0.003</td>
<td>0.01</td>
<td>0.04</td>
</tr>
</tbody>
</table>

© Performance Engineering Laboratory
**Future plans**

```cpp
template <
    typename element,
    typename allocator = std::allocator<element>,
    typename implementation =
        bounds_checked_vector<element, allocator>
>
class cphstl::vector {
    ...
}
```

**Possible std::vector implementations**

- bounds_checked_vector
- contiguous_vector
- iterator_safe_vector
- space_efficient_vector

**Possible std::deque implementations**

- bounds_checked_deque
- two_level_deque
- space_efficient_deque