Experiences with the design and implementation of space-efficient deques

Jyrki Katajainen
Bjarke Buur Mortensen

Datalogisk Institut
Københavns Universitet

5th Workshop on Algorithm Engineering
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
**Terminology: Resizable Arrays**

**Resizable arrays:** dynamic memory allocation; efficient sequence operations (push, pop and random access)

**Singly resizeable arrays:** modifying operations at one end (push_back, pop_back)

**Doubly resizeable arrays:** modifying operations at both ends

**Vector:** singly resizeable array, supporting general modifying operations (insert, erase).

**Deque:** doubly resizeable array, supporting general modifying operations
std::deque in the C++ library

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

- unbounded extra space
- $O(1)$ amortized time push operations
Earlier results

Resizable arrays [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]

- insert and erase operations in $O(n^\varepsilon)$ amortized time for any fixed constant $\varepsilon > 0$. Reference implementation has $\varepsilon = 1/2$

Deques [Mortensen, 2001]

- some implementation details were missing in [Brodnik et al., 1999]
- after filling in these details the implementation got complicated
Levelwise-allocated piles

- can be used to implement vector
- sequence operations in $O(1)$ worst-case time
- element with index $k \in [0 \ldots n-1]$ has index $k - 2^\lceil \log_2(k+1) \rceil + 1$ at level $\lceil \log_2(k+1) \rceil$
- $O(n)$ extra space
Blockwise-allocated piles

- # blocks at level $\ell$: $2^{\lceil \ell/2 \rceil}$
- Block size at level $\ell$: $2^{\lceil \ell/2 \rceil}$
- Sequence operations in $O(1)$ worst-case time
- $O(\sqrt{n})$ extra space
Faster modifying operations

[Goodrich and Kloss II, 1999]

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

- index of $i$th element in a block of size $b$: $(first + i) \mod b$
- each block in blockwise-allocated pile is a circular array
- modifying operations in $O(\sqrt{n})$ worst-case time
- in the twin-pile we have to store twice 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 and piecewise, and the structure can be used simultaneously during such a construction.

- sequence operations in $O(1)$ worst-case time
- modifying operations in $O(\sqrt{n})$ time
- $O(\sqrt{n})$ extra space
- restructuring only during pop operations
Experimental results

**Time measurements:** What is the cost of being space efficient?

**Space measurements:** How space efficient are we in practical terms?

**Setup:**
- Red Hat Linux 6.1
- 2 x Pentium III, 933 Mhz
- 1 GB RAM
- GCC 2.95.2
- SGI STL 3.3
## Time measurements

<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>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>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></td>
<td>10 000</td>
<td>100 000</td>
<td>1 000 000</td>
</tr>
<tr>
<td>std::deque</td>
<td>0.07</td>
<td>1.00</td>
<td>17.5</td>
</tr>
<tr>
<td>our deque</td>
<td>0.003</td>
<td>0.01</td>
<td>0.04</td>
</tr>
</tbody>
</table>
Space overhead measurement (after push_back)

std::deque
Our space-economical deque

overhead
ints

elements

overhead

0.0 %
0.5 %
1.0 %
1.5 %
2.0 %
2.5 %
3.0 %

0 10 M 20 M 30 M 40 M 50 M
Conclusions

Lessons learned

- use a deque instead of a vector to save space, but not necessarily a space-efficient one
- SGI’s fixed block size approach is good in practice but certain aspects can be improved

Future plans

- incorporate faster random access in `std::deque`
- provide multiple implementations of deque, e.g.
  - `space_efficient_deque<...>`
  - `deque<..., int blocksize>`
  - `insert_erase_deque<...>`