Århus, 28 August 2001

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

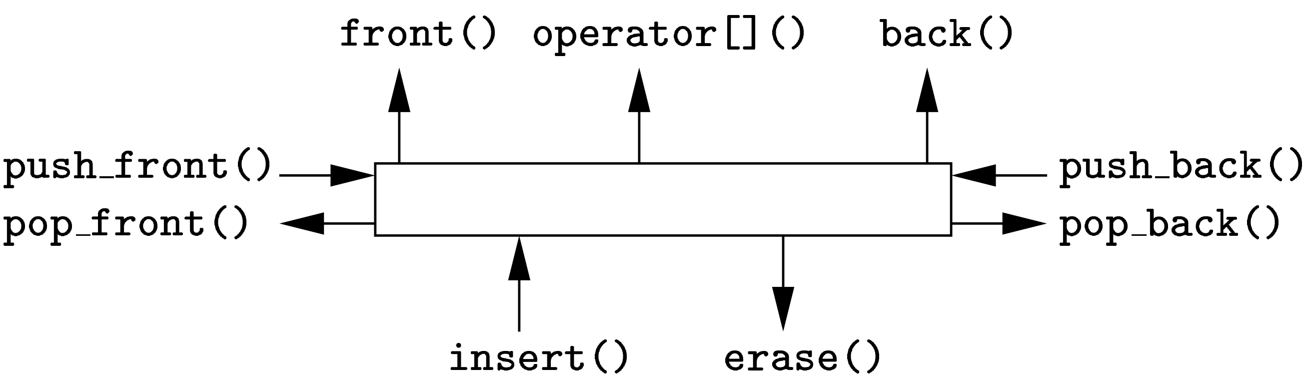
Resizable arrays: dynamic memory allocation; efficient sequence operations (push, pop and random access)

Singly resizable arrays: modifying operations at one end (`push_back`, `pop_back`)

Doubly resizable arrays: modifying operations at both ends

Vector: singly resizable array, supporting general modifying operations (`insert`, `erase`).

Deque: doubly resizable array, supporting general modifying operations

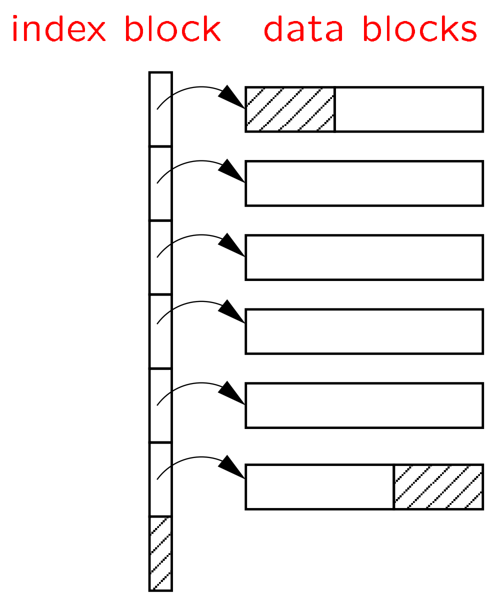


`std::deque` in the C++ library

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

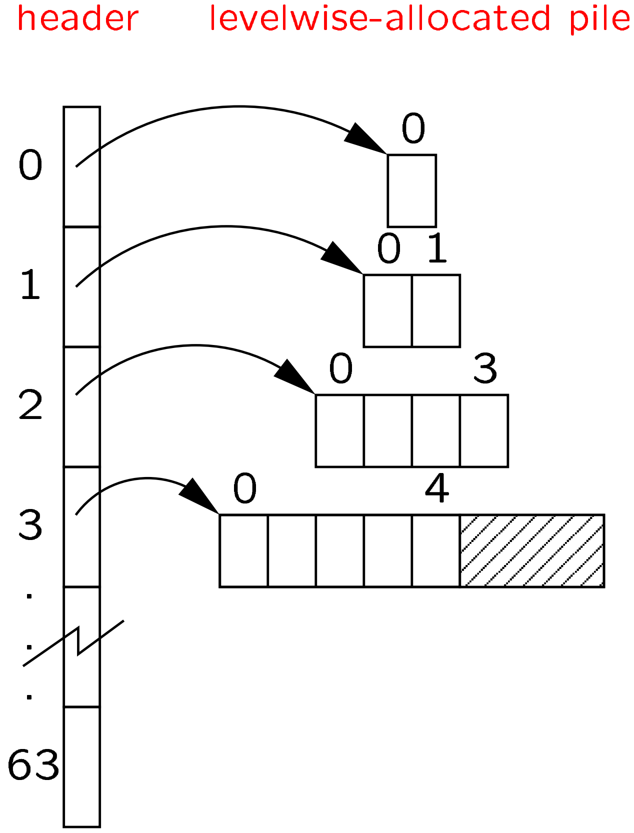
Vectors [Goodrich and Kloss II, 1999]

- 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]

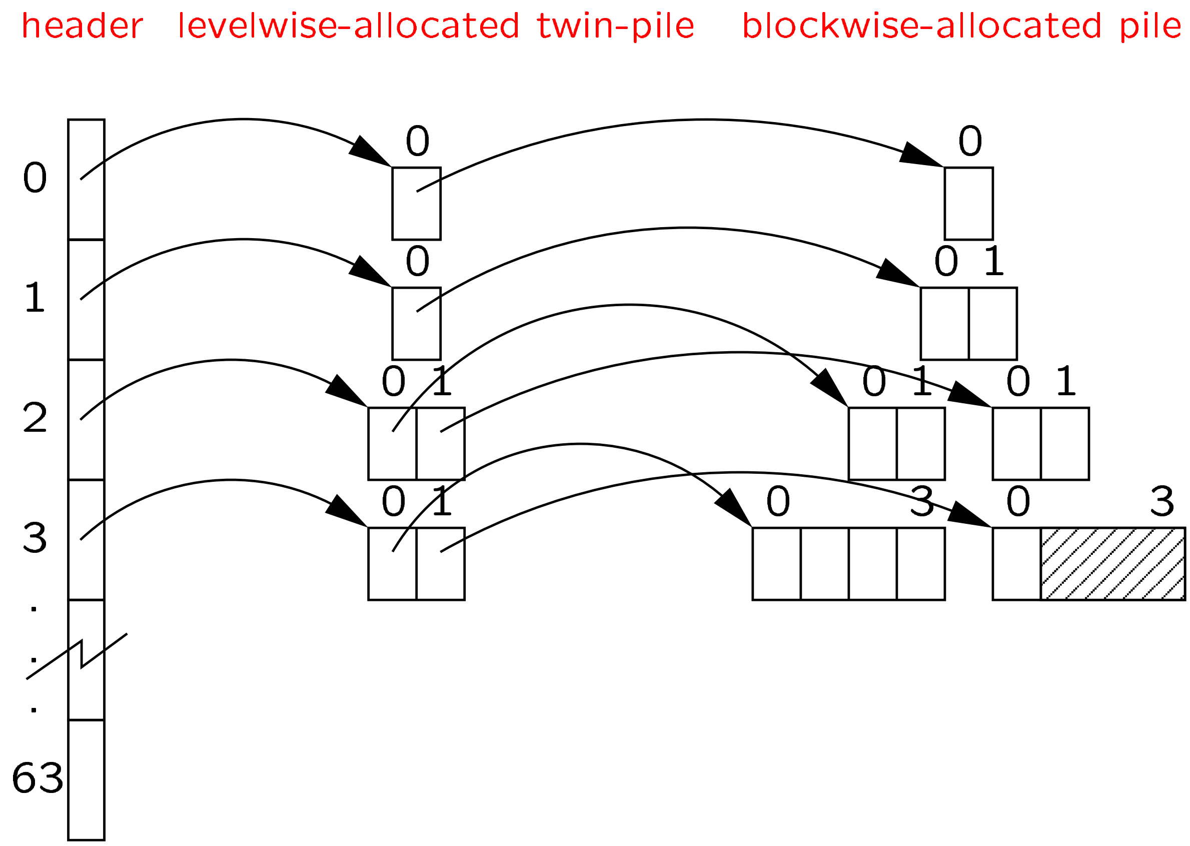
- some implementation details were missing in [Brodnik et al., 1999]
- after filling in these details the implementation got complicated

Levelwise-allocated piles



- can be used to implement vector
- sequence operations in $O(1)$ worst-case time
- element with index $k \in [0\,..\,n{-}1]$ has index $k - 2^{\lfloor \log_2(k+1) \rfloor} + 1$ at level $\lfloor \log_2(k+1) \rfloor$
- $O(n)$ extra space

Blockwise-allocated piles

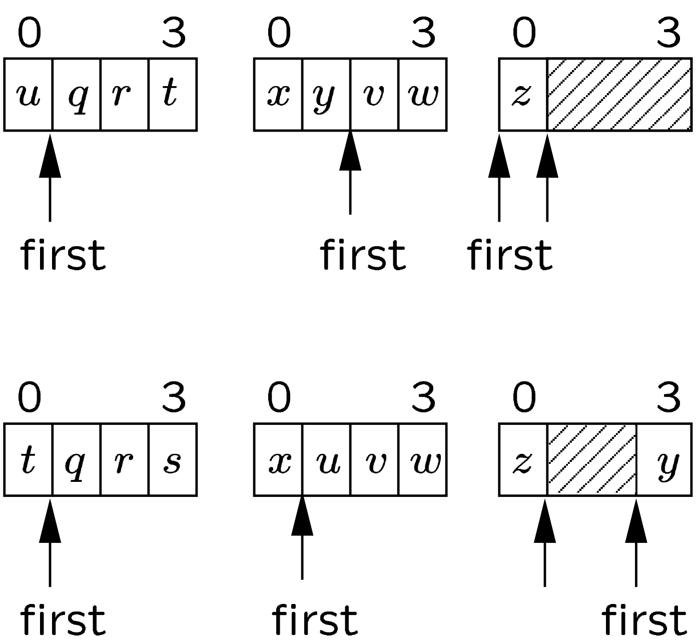


- # blocks at level ℓ: $2^{\lfloor \ell/2 \rfloor}$
- block size at level ℓ: $2^{\lceil \ell/2 \rceil}$
- sequence operations in $O(1)$ worst-case time
- $O(\sqrt{n})$ extra space

Faster modifying operations

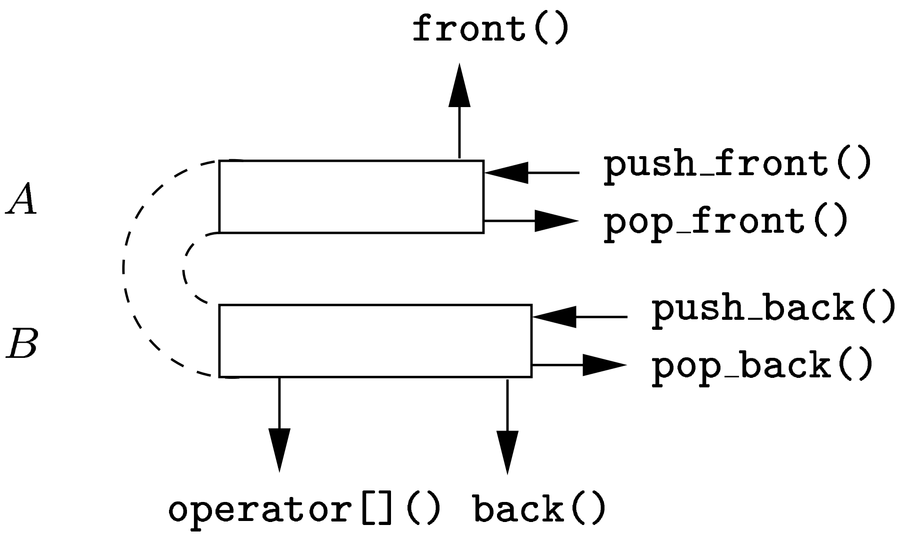
[Goodrich and Kloss II, 1999]

insert element s between r and t



- index of ith 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

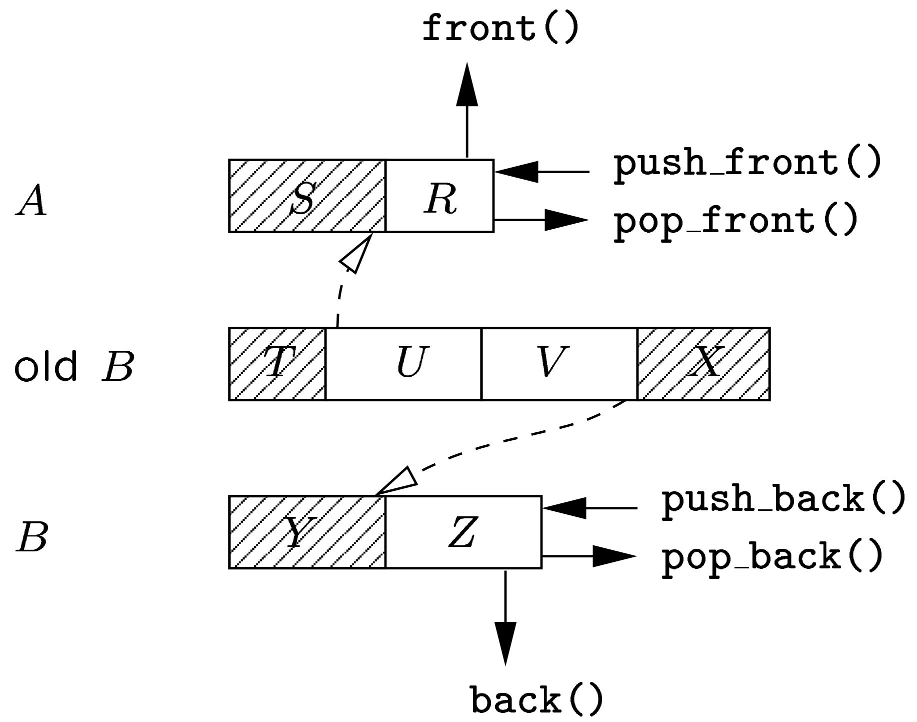
Space-efficient deques



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:

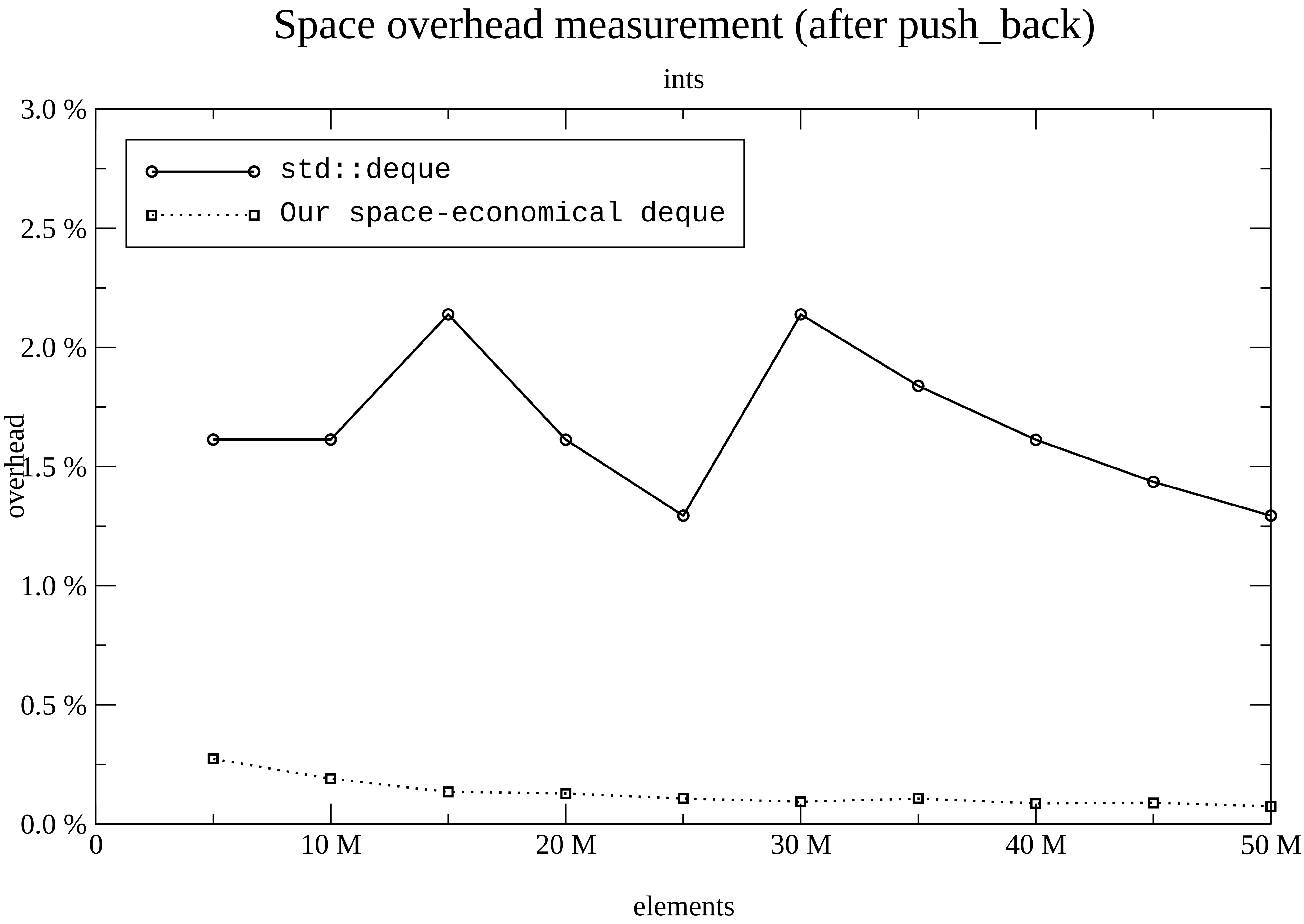
Red Hat Linux 6.1
2 x Pentium III, 933 Mhz
1 GB RAM
GCC 2.95.2
SGI STL 3.3

Time measurements

container	push_back (ns)	pop_back (ns)
std::deque	85	11
our deque	113	35
our deque (with reorganization)	113	375

container	sequential access (ns)	random access (ns)
std::deque	117	210
our deque	56	160
our deque (with reorganization)	58	162

container	1 000 inserts (s) initial size 10 000	1 000 inserts (s) initial size 100 000	1 000 inserts (s) initial size 1 000 000
std::deque	0.07	1.00	17.5
our deque	0.003	0.01	0.04


Space overhead measurement (after push_back)
ints
std::deque
Our space-economical deque
overhead
3.0 %
2.5 %
2.0 %
1.5 %
1.0 %
0.5 %
0.0 %
0
10 M
20 M
30 M
40 M
50 M
elements

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_eficient_deque<...>`
- `deque<..., int blocksize>`
- `insert_erase_deque<...>`