

4 April, 2016

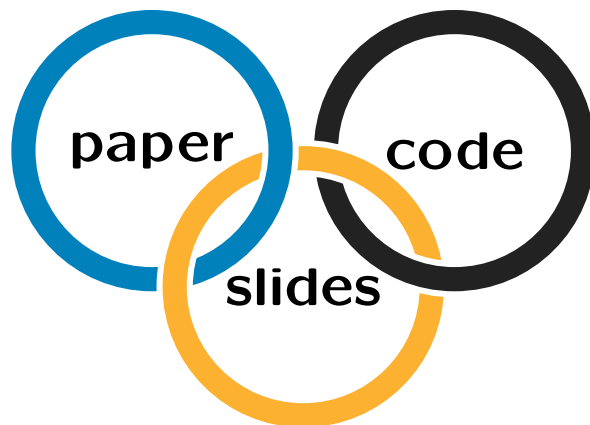
Last revision: 7 July, 2017

ARCO Meeting, Odense

Worst-case-efficient dynamic arrays in practice

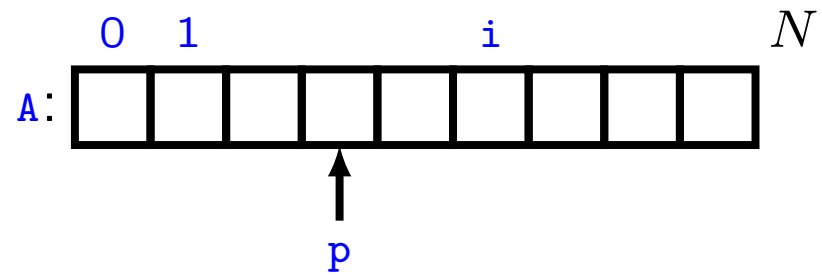
Jyrki Katajainen

Department of Computer Science
University of Copenhagen



goto my research information system
at <http://www.diku.dk/~jyrki/>

C array



- contiguous memory segment
- capacity fixed (denoted N)
- uninitialized
- no bounds checking

- **fast random access:**

$$A[i] \equiv *(A + i)$$

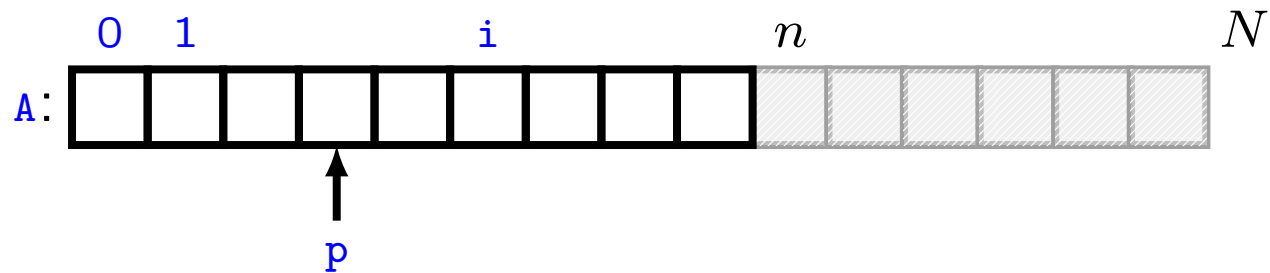
- **iteration:**

```
p = A;
```

```
++p;
```

```
*p;
```

C++ array



- contiguous memory segment
- size varies (`++`, `--`, denoted `n`)
- capacity varies (denoted `N`)
- initialized (up to `n`)
- bounds checking if desired
- **fast random access:**
`A[i] ≡ *(A.begin() + i)`
- **iteration:**
`p = A.begin();`
`++p;`
`*p;`

Array interfaces

$\mathcal{V}, \mathcal{A} = \text{std}::\text{allocator}<\mathcal{V}>$ std::vector
<ul style="list-style-type: none"> + \mathcal{V}: value type + \mathcal{A}: allocator type + \mathcal{I}: iterator type + \mathbb{N}: size type ...
<ul style="list-style-type: none"> + default constructor + destructor + get_allocator() const $\rightarrow \mathcal{A}$ + begin() const $\rightarrow \mathcal{I}$ + end() const $\rightarrow \mathcal{I}$ + size() const $\rightarrow \mathbb{N}$ + resize(\mathbb{N}) \rightarrow void + capacity() const $\rightarrow \mathbb{N}$ + reserve(\mathbb{N}) \rightarrow void + shrink_to_fit() \rightarrow void + operator[](\mathbb{N}) const $\rightarrow \mathcal{V}$ const& + operator[](\mathbb{N}) $\rightarrow \mathcal{V}$& + push_back(\mathcal{V} const&) \rightarrow void + push_back(\mathcal{V}&&) \rightarrow void + pop_back() \rightarrow void + insert(\mathcal{I}, \mathcal{V} const&) $\rightarrow \mathcal{I}$ + erase(\mathcal{I}) $\rightarrow \mathcal{I}$...

C++ standard

62 member functions
7 free functions

\mathcal{V} leda::array
<ul style="list-style-type: none"> + \mathcal{V}: value type + \mathcal{I}: iterator type + item: \mathcal{I} + \mathbb{N}: size type ...
<ul style="list-style-type: none"> + default constructor + copy constructor + copy assignment + destructor + first_item() $\rightarrow \mathcal{I}$ + last_item() $\rightarrow \mathcal{I}$ + next_item(\mathcal{I}) $\rightarrow \mathcal{I}$ + prev_item(\mathcal{I}) $\rightarrow \mathcal{I}$ + begin() $\rightarrow \mathcal{I}$ + end() $\rightarrow \mathcal{I}$ + low() const $\rightarrow \mathbb{N}$ + high() const $\rightarrow \mathbb{N}$ + size() const $\rightarrow \mathbb{N}$ + resize(\mathbb{N}) \rightarrow void + get(\mathbb{N}) $\rightarrow \mathcal{V}$& + set(\mathbb{N}, \mathcal{V}) \rightarrow void + operator[](\mathbb{N}) $\rightarrow \mathcal{V}$& ...

LEDA user manual

36 member functions

Bridge design pattern

$\mathcal{V}, \mathcal{A} = \text{std::allocator}\langle\mathcal{V}\rangle, \mathcal{K} = \text{std::vector}\langle\mathcal{V}, \mathcal{A}\rangle$

cphstl::vector
<ul style="list-style-type: none"> + \mathcal{V}: value type + \mathcal{A}: allocator type + \mathcal{K}: kernel type ...
...

$\mathcal{V}, \mathcal{K} = \text{std::vector}\langle\mathcal{V}, \text{std::allocator}\langle\mathcal{V}\rangle\rangle$

cphleda::array
<ul style="list-style-type: none"> + \mathcal{V}: value type + \mathcal{K}: kernel type ...
...

$\mathcal{V}, \mathcal{A} = \text{std::allocator}\langle\mathcal{V}\rangle$

array kernel
<ul style="list-style-type: none"> + \mathcal{V}: value type + \mathcal{A}: allocator type + \mathcal{K}: array kernel$\langle\mathcal{V}, \mathcal{A}\rangle$ + \mathcal{I}: rank iterator$\langle\mathcal{K}\rangle$ + \mathcal{J}: rank iterator$\langle\mathcal{K} \text{ const}\rangle$ + \mathbb{N}: size type - \mathcal{Args}: any argument-pack type ...
<ul style="list-style-type: none"> - <code>index_to_address(\mathbb{N}) const</code> $\rightarrow \mathcal{V}^*$ + default constructor + destructor + <code>get_allocator() const</code> $\rightarrow \mathcal{A}$ + <code>swap($\mathcal{K}\&$)</code> \rightarrow void + <code>begin() const</code> $\rightarrow \mathcal{I}$ + <code>end() const</code> $\rightarrow \mathcal{I}$ + <code>size() const</code> $\rightarrow \mathbb{N}$ + <code>capacity() const</code> $\rightarrow \mathbb{N}$ + <code>operator[](\mathbb{N}) const</code> $\rightarrow \mathcal{V} \text{ const}\&$ + <code>operator[](\mathbb{N})</code> $\rightarrow \mathcal{V}\&$ + <code>emplace_back($\mathcal{Args}\&\&\dots$)</code> \rightarrow void + <code>push_back($\mathcal{V} \text{ const}\&$)</code> \rightarrow void + <code>push_back($\mathcal{V}\&\&$)</code> \rightarrow void + <code>pop_back()</code> \rightarrow void

minimal

Usage example

```
#include <algorithm> // std::sort
#include <cassert> // assert
#include <iostream> // standard streams
#include <memory> // std::allocator
#include "sliced_array.h++" // cphstl::sliced_array
#include "vector.h++" // cphstl::vector

int main(int, char**) {
    using V = int;
    using A = std::allocator<V>;
    using S = cphstl::vector<V, A, cphstl::sliced_array<V, A>>;

    S sequence = {4, 2, 3, 1};
    std::sort(sequence.begin(), sequence.end());
    assert(std::is_sorted(sequence.begin(), sequence.end()));
    for (V x : sequence) {
        std::cout << x << " ";
    }
    std::cout << "\n";
    return 0;
}
```

```
jyrki@Janus:~/CPHSTL/Paper/Dynamic-arrays/Test$ make usage
g++ -std=c++11 -Wall -x c++ -I../Code usage.c++
./a.out
1 2 3 4
```

Reflection-based implementation

```
template <typename  $\mathcal{T}$ >
class has_shrink_to_fit {

    template <typename  $\mathcal{U}$ , void ( $\mathcal{U}::*$ )()>
    struct check;

    template <typename  $\mathcal{U}$ >
    static char member(check< $\mathcal{U}$ , & $\mathcal{U}::$ shrink_to_fit>*);

    template <typename  $\mathcal{U}$ >
    static int member(...);

public:

    enum { value = sizeof(member< $\mathcal{T}\mathcal{V}$ , typename  $\mathcal{A}$ , typename  $\mathcal{K}$ >
void vector< $\mathcal{V}$ ,  $\mathcal{A}$ ,  $\mathcal{K}$ >::shrink_to_fit() {
    shrink_to_fit_dispatch(typename std::conditional<has_shrink_to_fit< $\mathcal{K}$ >:: ←
        value, std::true_type, std::false_type>::type());
}

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

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

Goal

In a minimal kernel, support all operations at $O(1)$ worst-case cost.

Assumptions: For allocators a , b , and integer $\Delta \geq 0$:

- $p = a.\text{allocate}(\Delta)$: $O(1)$ worst case for any Δ
- $a.\text{deallocate}(p, \Delta)$: $O(1)$ worst case for any Δ
- $a.\text{construct}(p, \bullet)$: $O(1)$ worst case
- $a.\text{destroy}(p)$: $O(1)$ worst case
- $a = b$: $O(1)$ worst case
- word-RAM primitives: $O(1)$ worst case

Memory-management tests

1. Repeat t times ($t = 10^6$):
 - a) Allocate space for a block of Δ bytes ($\Delta = 2^k$).
 - b) Deallocate the allocated block without using it.
2. Repeat the above r times ($r = 101$) and report the median of the execution times for a single allocate-deallocate pair.

memory-management tests; median of the execution times [ns]

```
std::allocator<char> a; a.deallocate(a.allocate( $\Delta$ ), $\Delta$ );
```

2^5	2^6	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}	2^{22}	2^{23}	2^{24}	2^{25}	2^{26}
-------	-------	-------	-------	-------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------

35	35	61	61	60	67	68	66	66	68	68	68	50	51	52	52	52	51	51	52	2456	2644
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	------	------

```
free(malloc( $\Delta$ ));
```

2^5	2^6	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}	2^{22}	2^{23}	2^{24}	2^{25}	2^{26}
-------	-------	-------	-------	-------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------

27	27	49	50	48	57	57	55	55	57	57	57	37	39	39	39	39	39	39	39	39	39	2360	2366
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	------	------

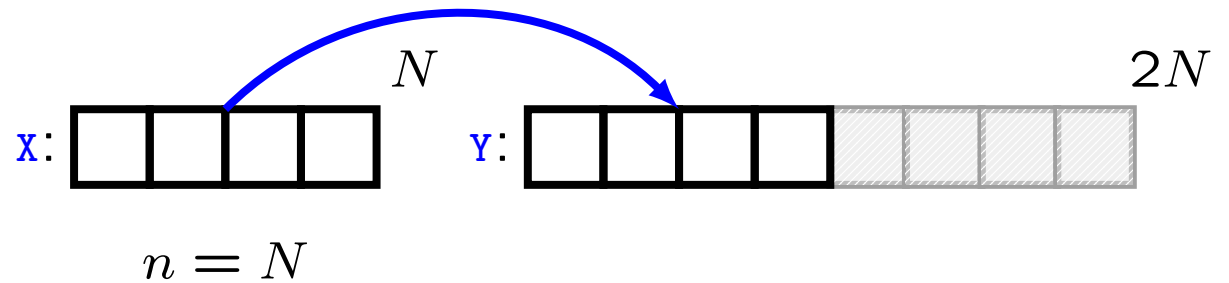
I know, I may have a problem, but you have it too!

Textbook solution

• doubling

if ($n = N$)

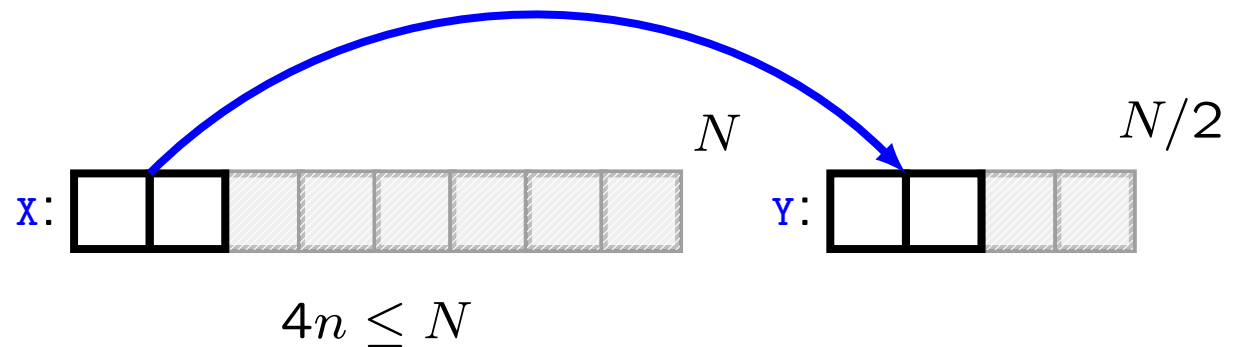
1. allocate $2N$
2. move all from x to y
3. release x



• halving

if ($4n \leq N$)

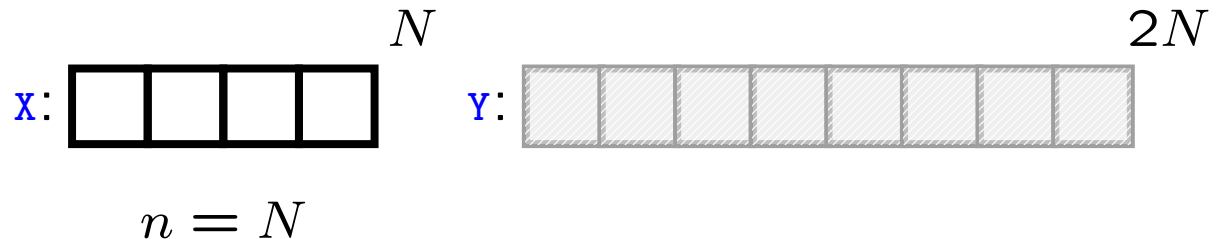
1. allocate $N/2$
2. move all from x to y
3. release x



Folklore solution (`cphstl :: resizable_array`)

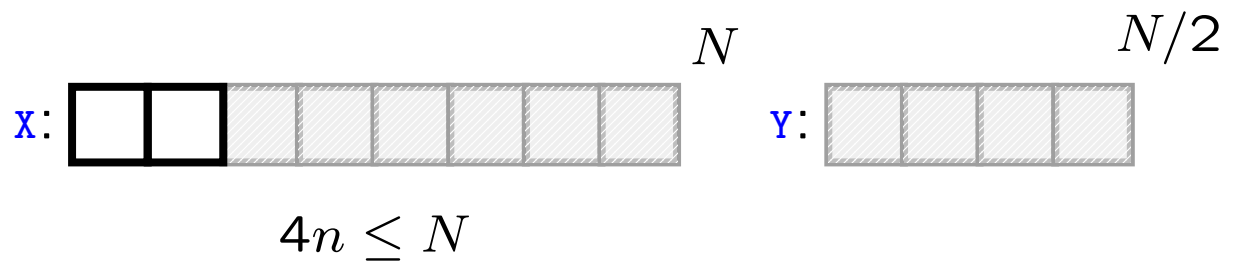
- **doubling**

```
if (n = N)
  allocate 2N
```



- **halving**

```
if (4n ≤ N)
  allocate N/2
```



- **incremental copying**

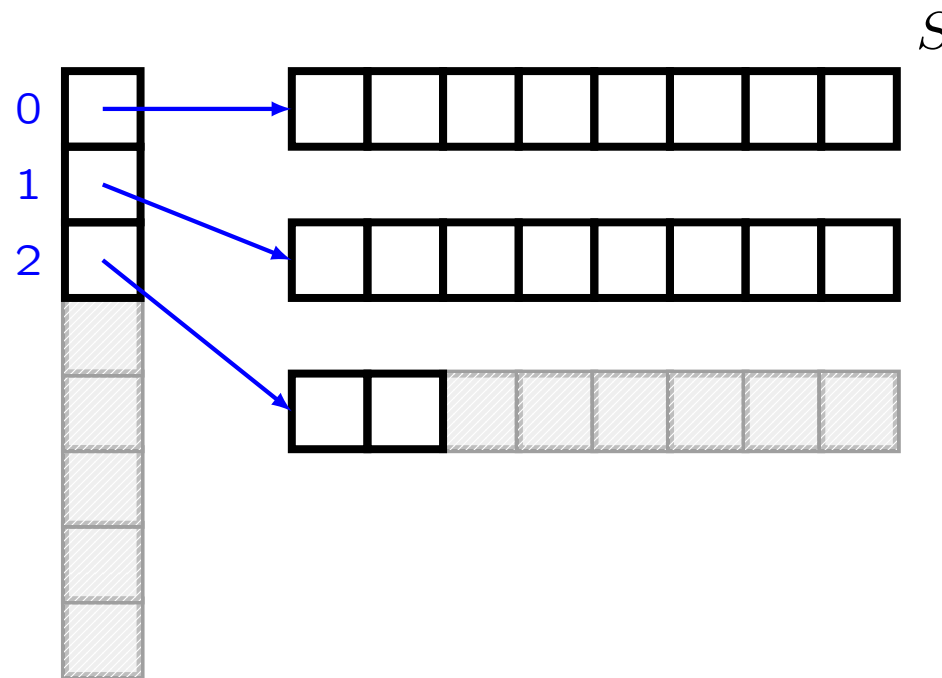
`push_back`: move 1 from the end of `x` to `y` at the same relative position

`pop_back`: move 2 from the end of `x` to `y` at the same relative position

```
if (x empty)
  release x
```

Slicing (`cphstl :: sliced_array`)

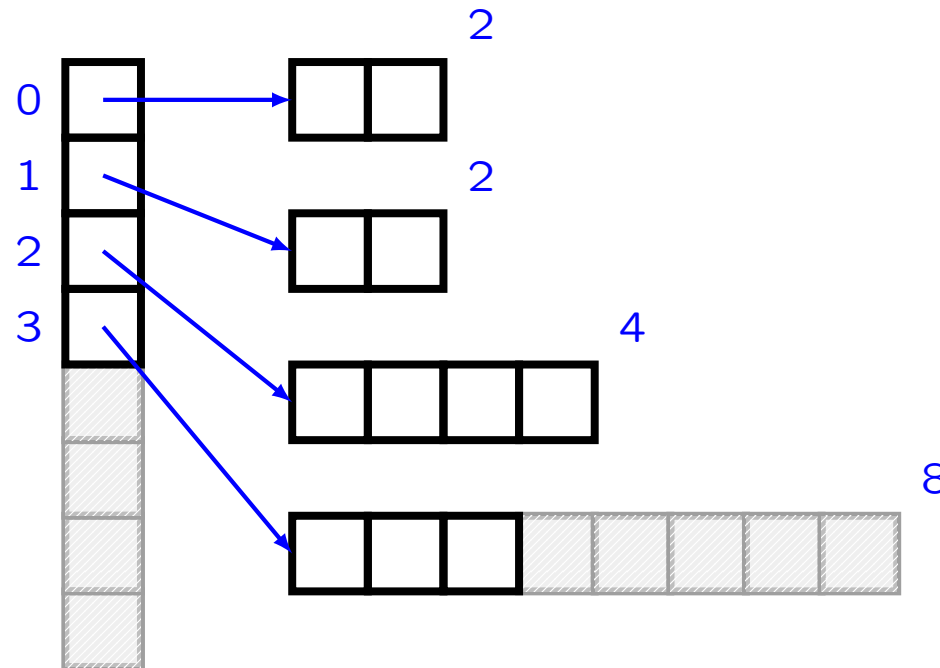
- S : slice capacity (specified at compile time);
- **slice**: C array of capacity S ;
- **# slices**: $\lceil n/S \rceil$;
- **last slice**: can be non-full;
- **directory**: resizable array



- **extra space**: at most S elements plus $O(n/S)$ pointers

Piling (`cphstl :: pile`)

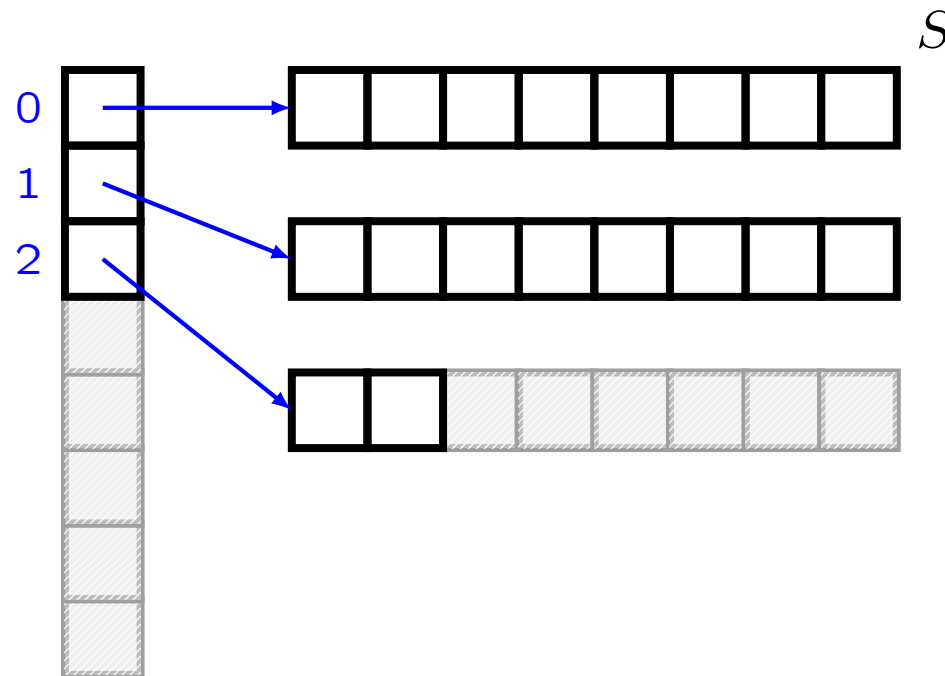
- **slice**: C array of fixed capacity (specified at run time);
- slice capacities exponentially increasing;
- **# slices**: $\lceil \lg(\max\{2, n\}) \rceil$;
- **last slice**: can be non-full;
- **directory**: resizable array



- **extra space**: at most n elements plus $O(\lg n)$ pointers

Hashed array tree

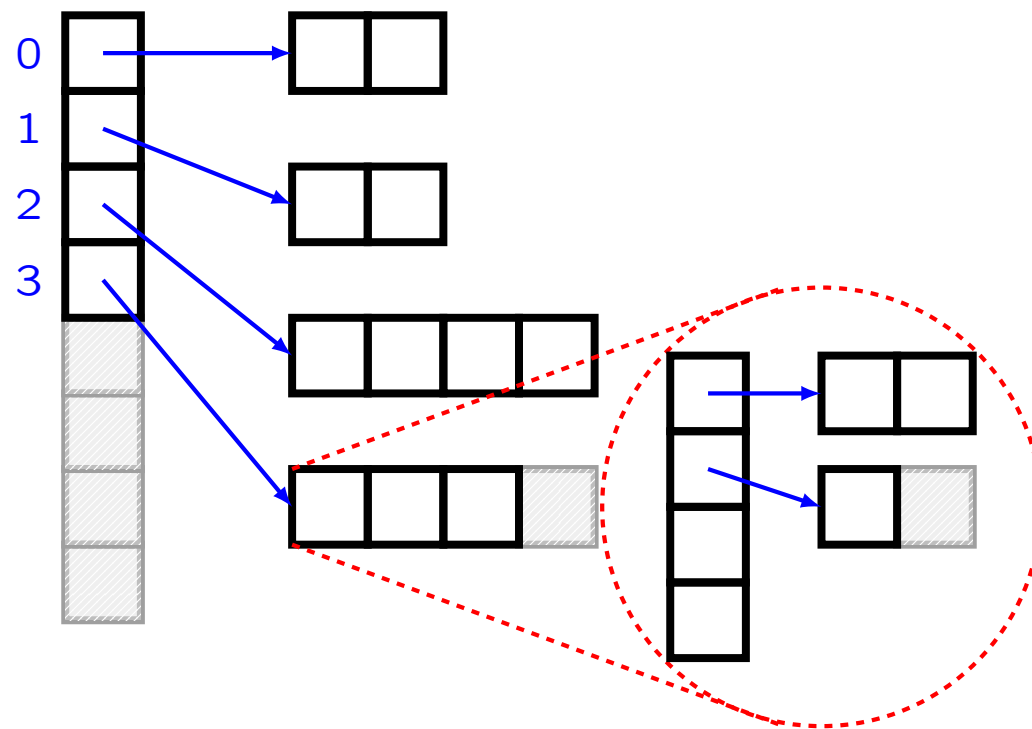
- N : fixed capacity (given at run time);
- $S = 2^{\lceil \lg N/2 \rceil} \in (\sqrt{N}/2 .. \sqrt{N}]$;
- **slice**: C array of capacity S ;
- **# slices**: $\lceil n/S \rceil$;
- **last slice**: can be non-full;
- **directory**: C array of capacity $\lceil N/S \rceil$



- **extra space**: at most \sqrt{N} elements plus $O(\sqrt{N})$ pointers

Piling hashed array trees (`cphst1 :: space_efficient_array`)

- pile of hashed array trees; • **at level i** : hashed array tree of capacity $\max\{2, 2^i\}$; • **# levels**: $\lceil \lg(\max\{2, n\}) \rceil$; • **last slice in the last hashed array tree**: can be non-full; • **directory**: resizable array



- **extra space**: at most \sqrt{n} elements plus $O(\sqrt{n})$ pointers; space bound $\Omega(\sqrt{n})$ optimal

Summary of efficiencies

- S : slice size; • n : current size

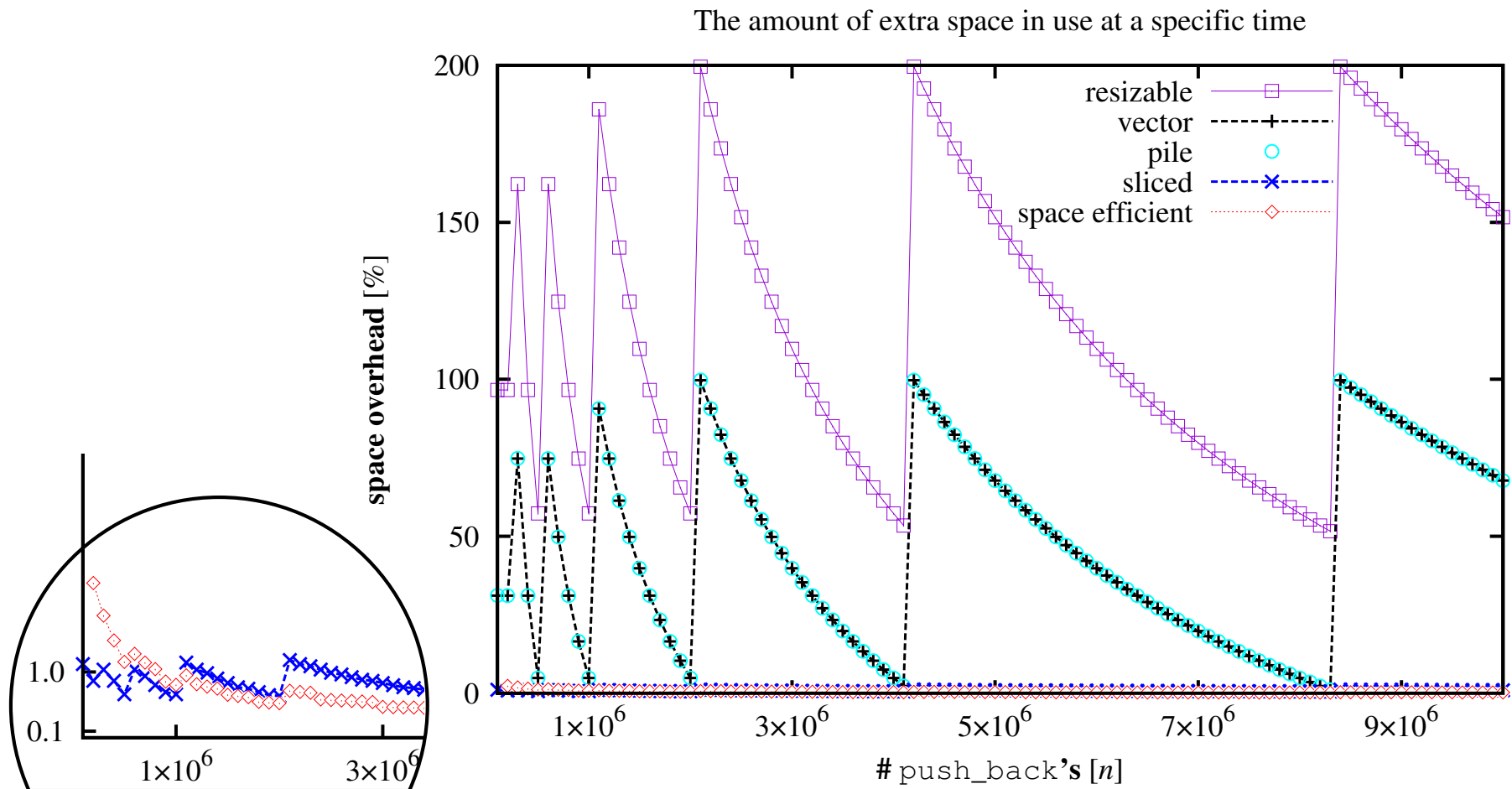
solution	operator []	push_back pop_back	elements	pointers
textbook	$O(1)$	$O(1)$ amortized	$6n$	$O(1)$
resizable	$O(1)$	$O(1)$	$12n$	$O(1)$
sliced	$O(1)$	$O(1)$	$n + S$	$O(n/S)$
pile	$O(1)$	$O(1)$	$2n$	$O(\lg n)$
space efficient	$O(1)$	$O(1)$	$n + \sqrt{n}$	$O(\sqrt{n})$

- **sources:** [Sitarski 1996]; [Brodnik, Carlsson, Demaine, Munro, Sedgewick 1999]; [Katajainen, Mortensen 2001]

Space test

- **overhead:** $100\% \cdot (\text{space in use} - n \cdot \text{sizeof}(\text{int})) / n \cdot \text{sizeof}(\text{int})$

space overhead after n `push_back` operations



Subscripting operator

```
 $\forall$ & operator[]( $\mathbb{N}$  i) {  
    return *index_to_address(i);  
}
```

contiguous array

```
 $\forall$ * index_to_address( $\mathbb{N}$  i) const {  
    return A + i;  
}
```

resizable array

```
 $\forall$ * index_to_address( $\mathbb{N}$  i) const {  
    if (i < X_size) {  
        return X + i;  
    }  
    return Y + i;  
}
```

pile

```
 $\mathbb{N}$  whole_number_logarithm( $\mathbb{N}$  x) {  
    asm("bsr %0, %0\n"  
        : "=r"(x)  
        : "0" (x)  
    );  
    return x;  
}
```

```
 $\forall$ * index_to_address( $\mathbb{N}$  i) const {  
    if (i < 2) {  
        return directory[0] + i;  
    }  
     $\mathbb{N}$  h = whole_number_logarithm(i);  
    return directory[h] + i - (1 << h);  
}
```

sliced array

```
 $\forall$ * index_to_address( $\mathbb{N}$  i) const {  
    return directory[i >> shift] + (i bitand mask);  
}
```

space-efficient array

```
 $\forall$ * index_to_address( $\mathbb{N}$  i) const {  
    if (i < 2) {  
        return directory[0].index_to_address(i);  
    }  
     $\mathbb{N}$  h = whole_number_logarithm(i);  
     $\mathbb{N}$   $\Delta$  = i - (1 << h);  
    return directory[h].index_to_address( $\Delta$ );  
}
```

Sorting tests

- **computer:** my Linux box; • **compiler:** `g++ -O3`; • **data:** `int`

introsort tests; running time per $n \lg n$ [ns]

n	vector	resizable	pile	sliced	space efficient
2^{10}	3.56	6.18	9.31	8.35	12.0
2^{15}	3.56	5.96	8.99	8.05	11.6
2^{20}	3.48	5.84	8.80	7.91	11.3
2^{25}	3.48	5.79	8.67	7.80	11.2

heapsort tests; running time per $n \lg n$ [ns]

n	vector	resizable	pile	sliced	space efficient
2^{10}	4.83	8.89	17.1	12.5	20.3
2^{15}	4.94	8.47	16.6	12.3	19.8
2^{20}	7.18	10.7	17.8	15.7	21.8
2^{25}	23.5	27.7	33.3	37.0	39.8

Modification tests

n `push_back` operations; running time per n [ns]

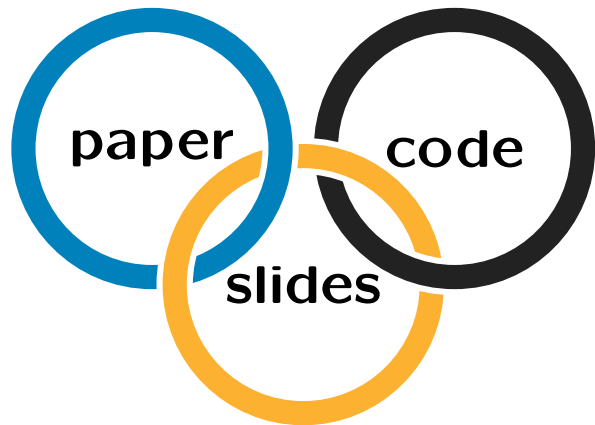
n	vector	resizable	pile	sliced	space efficient
2^{10}	4.23	5.18	5.65	4.65	10.3
2^{15}	3.52	6.39	5.16	4.63	7.35
2^{20}	4.78	8.48	5.12	4.60	6.92
2^{25}	4.15	8.42	4.55	4.58	6.75

n `pop_back` operations; running time per n [ns]

n	vector	resizable	pile	sliced	space efficient
2^{10}	0.0	3.62	3.08	2.56	8.15
2^{15}	0.0	2.99	2.15	2.60	5.55
2^{20}	0.0	2.86	2.27	2.41	5.17
2^{25}	0.0	2.91	2.11	2.43	5.07

Conclusions

- **worst-case efficiency:** an array cannot be a contiguous memory segment
- **insert, erase:** unnatural operations in this context
- **reflection-based implementation:** zero overhead
- **small kernels:** adaptability can be provided relatively cheaply.



goto my research information system
at <http://www.diku.dk/~jyrki/>