

# Algorithms and Datastructures

## Static Arrays, Dynamic Arrays, Amortized Analysis

Albert-Ludwigs-Universität Freiburg



**UNI  
FREIBURG**

Prof. Dr. Rolf Backofen

Bioinformatics Group / Department of Computer Science  
Algorithms and Datastructures, December 2017

## Static Arrays

## Dynamic Arrays

Introduction

Amortized Analysis

- Static arrays exist in nearly every programming language
- They are initialized with a fixed size  $n$
- **Problem:** The needed size is not always clear at compile time

Table: Static array with size  $n = 5$

| Index | 0   | 1   | 2   | 3   | 4   |
|-------|-----|-----|-----|-----|-----|
| Value | "a" | "b" | "c" | "d" | "e" |

### Python:

- We have dynamic sized lists
- Python does automatic resizing when needed

```
# Creates a list of "0"s with init. size 10
numbers = [0] * 10
```

```
# Prints number at index 7 ("0")
print("%d" % numbers[7])
```

```
# Saves number 42 at index 8
numbers[8] = 42
```

```
# Prints the number at index 8 ("42")
print("%d" % numbers[8])
```

- The name “static array” has nothing to do with the keyword **static** from Java / C++
- Nor is the array allocated before the program starts
- The **size** of the array is static and can not be changed after creation
- The name “fixed-size array” would be more appropriate

### Dynamic arrays:

- The array is created with an initial size
- The size can be dynamically modified
- **Problem:** We need a dynamic structure to store the data

### Python:

```
greetings = ["Good morning", "ohai"]

greetings.append("Guten morgen")
greetings.append("bonjour")

# Prints text at index 2 ("Guten morgen")
print("%s" % greetings[2])

# Removes all elements
greetings.clear();
```

- We store the data in a fixed-size array with the needed size
- **Append:**
  - Create fixed-size array with the needed size
  - Copy elements from the old to the new array
- **Remove:**
  - Create fixed-size array with the needed size
  - Copy elements from the old to the new array



First implementation:

- We resize the array before each append
- We choose the size exactly as needed

```
class DynamicArray:

    def __init__(self):
        self.size = 0
        self.elements = []

    def capacity(self):
        return len(self.elements)

    ...
```

```
class DynamicArray:
    ...

    def append(self, item):
        newElements = [0] * (self.size + 1)

        for i in range(0, self.size):
            newElements[i] = self.elements[i]

        self.elements = newElements




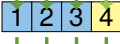

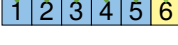
        newElements[self.size] = item
        self.size += 1
```

- Why is the runtime quadratic?



Figure: Runtime of *DynamicArray*

### Runtime:

|  |            |                                  |
|--|------------|----------------------------------|
|   | $O(1)$     | write 1 element                  |
|  | $O(1 + 1)$ | write 1 element, copy 1 element  |
|  | $O(1 + 2)$ | write 1 element, copy 2 elements |
|  | $O(1 + 3)$ | write 1 element, copy 3 elements |
|  | $O(1 + 4)$ | write 1 element, copy 4 elements |
|  | $O(1 + 5)$ | write 1 element, copy 5 elements |
| ...  | ...        | ...                              |

### Analysis:

- Let  $T(n)$  be the runtime of  $n$  sequential append operations
- Let  $T_i$  be the runtime of each  $i$ -th operation
  - Then  $T_i = A \cdot i$  for a constant  $A$
  - We have to copy  $i - 1$  elements

$$\begin{aligned} T(n) &= \sum_{i=1}^n T_i = \sum_{i=1}^n (A \cdot i) = A \cdot \sum_{i=1}^n i = A \cdot \frac{n^2 + n}{2} \\ &= O(n^2) \end{aligned}$$

### Idea:

- Better resize strategy
- We allocate more space than needed
- We over-allocate a constant amount of elements
  - Amount:  $C = 3$  or  $C = 100$

```
def append(self, item):  
    if self.size >= len(self.elements):  
        newElements = [0] * (self.size + 100)  
  
        for i in range(0, self.size - 1):  
            newElements[i] = self.elements[i]  
  
        self.elements = newElements  
  
    self.elements[self.size] = item  
    self.size += 1
```



- Why is the runtime still quadratic?



Figure: Runtime of *DynamicArray*

### Runtime for $C = 3$ :

|  |            |                                  |
|--|------------|----------------------------------|
|  | $O(1)$     | write 1 element                  |
|  | $O(1)$     | write 1 element                  |
|  | $O(1)$     | write 1 element                  |
|  | $O(1 + 3)$ | write 1 element, copy 3 elements |
|  | $O(1)$     | write 1 element                  |
|  | $O(1)$     | write 1 element                  |
|  | $O(1 + 6)$ | write 1 element, copy 6 elements |
| ...  | ...        | ...                              |

### Analysis:

- Most of the append operations now just cost  $O(1)$
- Every  $C$  steps the costs for copying are added:  
 $C, 2 \cdot C, 3 \cdot C, \dots$  this means:

$$\begin{aligned}T(n) &= \sum_{i=1}^n A \cdot 1 + \sum_{i=1}^{n/C} A \cdot i \cdot C \\&= A \cdot n + A \cdot C \cdot \sum_{i=1}^{n/C} i \\&= A \cdot n + A \cdot C \cdot \frac{\frac{n^2}{C^2} + \frac{n}{C}}{2} \\&= A \cdot n + \frac{A}{2 \cdot C} \cdot n^2 + \frac{A}{2} \cdot n = O(n^2)\end{aligned}$$

- The factor of  $n^2$  is getting smaller

### Idea:

- Double the size of the array

```
def append(self, item):  
    if self.size >= len(self.elements):  
        newElements = [0] \  
            * max(1, 2 * self.size)  
  
        for i in range(0, self.size):  
            newElements[i] = self.elements[i]  
  
        self.elements = newElements  
  
    self.elements[self.size] = item  
    self.size += 1
```

- Now the runtime is linear with some bumps. Why?

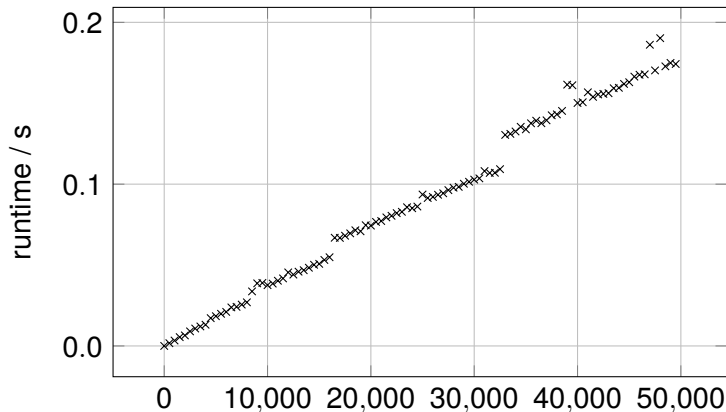


Figure: Runtime of *DynamicArray*

### Runtime for $C = 2$ (Double the size):

|  |            |                          |
|--|------------|--------------------------|
|   | $O(1)$     | write 1                  |
|  | $O(1 + 1)$ | write 1, copy 1 element  |
|  | $O(1 + 2)$ | write 1, copy 2 elements |
|  | $O(1)$     | write 1                  |
|  | $O(1 + 4)$ | write 1, copy 4 elements |
|  | $O(1)$     | write 1                  |
|  | $O(1)$     | write 1                  |
|  | $O(1)$     | write 1                  |
|  | $O(1 + 8)$ | write 1, copy 8 elements |
| ...  | ...        | ...                      |

### Analysis:

- Now all appends cost  $O(1)$
- Every  $2^i$  steps we have to add the cost  $A \cdot 2^i$  (for  $i = 0, 1, 2, \dots, k$  with  $k = \text{floor}(\log_2(n-1))$ )
- In total that accounts to:

$$\begin{aligned}T(n) &= n \cdot A + A \cdot \sum_{i=0}^k 2^i = n \cdot A + A(2^{k+1} - 1) \\&\leq n \cdot A + A \cdot 2^{(k+1)} \\&= n \cdot A + 2 \cdot A \cdot 2^{(k)} \\&\leq n \cdot A + 2 \cdot A \cdot n \\&= 3 \cdot A \cdot n \\&= O(n)\end{aligned}$$

### How do we shrink the array?

- If the array is half-full, we can shrink it by half, like for the append operation
  - If we *append* directly after *shrinking* we have to extend the array again
    - We leave some space for following append operations
- ⇒ We only shrink the array to 75%



### Analysis:

- **Difficult:** We have a random number of *append* / *remove* operations
- We can not exactly predict when resizing is happening



Figure: Static array with capacity  $c_i$

### Notation:

- We have  $n$  instructions  $O = \{O_1, \dots, O_n\}$
- The **size** after operation  $i$  is  $s_i$ , with  $s_0 := 0$
- The **capacity** after operation  $i$  is  $c_i$ , with  $c_0 := 0$
- The **cost** of operation  $i$  is  $\text{cost}(O_i)$  (previously named  $T_i$ )

Reallocation:  $\text{cost}(O_i) \leq A \cdot s_i$ ,

Insert / Delete (Update):  $\text{cost}(O_i) \leq A$ ,

# Dynamic Arrays

## Amortized Analysis - Example



| Operation |        |          | Size $s_i$ | Capacity $c_i$ | Costs<br>$\text{cost}(O_i)$ |
|-----------|--------|----------|------------|----------------|-----------------------------|
| $O_1$     | append | realloc. | $s_1$      | $c_1$          | $A \cdot s_1$               |
| $O_2$     | append |          | $s_2$      | $c_2 = c_1$    | $A \cdot 1$                 |
| $O_3$     | append |          | $s_3$      | $c_3 = c_1$    | $A \cdot 1$                 |
| $O_4$     | remove |          | $s_4$      | $c_4 = c_1$    | $A \cdot 1$                 |
| $O_5$     | remove | realloc. | $s_5$      | $c_5$          | $A \cdot s_5$               |
| $O_6$     | append |          | $s_6$      | $c_6 = c_5$    | $A \cdot 1$                 |
| $O_7$     | remove |          | $s_7$      | $c_7 = c_5$    | $A \cdot 1$                 |
| $O_8$     | append |          | $s_8$      | $c_8 = c_5$    | $A \cdot 1$                 |
| $O_9$     | append | realloc. | $s_9$      | $c_9$          | $A \cdot s_9$               |
| ...       | ...    |          | ...        | ...            | ...                         |
| $O_n$     | append |          | $s_n$      | $c_n$          | $A \cdot 1$                 |

### Implementation:

- If  $O_i$  is an *append* operation and  $s_{i-1} = c_{i-1}$ :
  - $\Rightarrow$  Resize array to  $c_i = \left\lfloor \frac{3}{2} s_i \right\rfloor = \text{floor} \left( \frac{3}{2} s_i \right)$
  - $\Rightarrow \text{cost}(O_i) = A \cdot s_i$



Figure: *Append* operation with reallocation

**Result:** after operation we have  $c_i = \frac{3}{2} \cdot s_i$

### Implementation:

- If  $O_i$  is an *remove* operation and  $s_{i-1} \leq \frac{1}{3}c_{i-1}$ :  
 $\Rightarrow$  Resize array to  $c_i = \left\lfloor \frac{3}{2}s_i \right\rfloor = \text{floor}\left(\frac{3}{2}s_i\right)$   
 $\Rightarrow \text{cost}(O_i) = A \cdot s_i$



Figure: Remove operation with reallocation

**Result:** after operation we have again  $c_i = \frac{3}{2} \cdot s_i$

### Idea for proof:

- Expensive are only operations where reallocations are necessary
- If we just reallocated, it takes some time until the next reallocation is required.
- **Assumption:** After a costly *reallocation* of size  $X$  we have at least  $X$  operations of runtime  $O(1)$
- **Then:** Total cost of  $n$  operations is maximally  $2 \cdot n$

Table: Dynamic Array with  $C_{\text{ext}} = \frac{3}{2}$

| Operation |      |          | Size<br>$s_i$ | Capacity<br>$c_i$                                      | Costs<br>$\text{cost}(O_i)$ |   |
|-----------|------|----------|---------------|--|-----------------------------|---|
| $O_1$     | app. | realloc. | $s_1$         | $c_1 = 4$  | $C_1 \cdot s_1$             | $\left\{ \begin{array}{l} \text{distance} \\ 4 \geq \left\lfloor \frac{s_1}{2} \right\rfloor \end{array} \right.$ |
| $O_2$     | app. |          | $s_2$         | $c_2 = c_1$  | $C_2$                       |   |
| $O_3$     | app. |          | $s_3$         | $c_3 = c_1$  | $C_2$                       |   |
| $O_4$     | app. |          | $s_4$         | $c_4 = c_1$  | $C_2$                       |   |
| $O_5$     | app. | realloc. | $s_5$         | $c_5 = \left\lfloor \frac{3}{2} s_5 \right\rfloor = 7$ | $C_1 \cdot s_5$             | $\left\{ \begin{array}{l} \text{distance} \\ 3 \geq \left\lfloor \frac{s_5}{2} \right\rfloor \end{array} \right.$ |
| $O_6$     | app. |          | $s_6$         | $c_6 = c_5$  | $C_2$                       |   |
| $O_7$     | app. |          | $s_7$         | $c_7 = c_5$  | $C_2$                       |   |
| $O_8$     | app. | realloc. | $s_8$         | $c_8 = \frac{3}{2} s_8 = 12$                           | $C_1 \cdot s_8$             |   |
| ...       | ...  | ...      | ...           | ...  | ...                         |   |

### To show:

- **Lemma:** If a *reallocation* occurs at  $O_i$  the nearest *reallocation* is at  $O_j$  with  $j - i > \frac{s_i}{2}$
- **Corollary:**  $\text{cost}(O_1) + \dots + \text{cost}(O_n) \leq 4A \cdot n$



# Dynamic Arrays

Proof: Worst Case Same Operation

Table: Case 1:  $\frac{1}{2}s_j$  appends

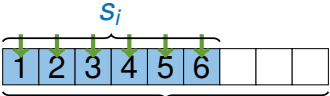




Array

Costs

|             |   |  |
|-------------|---|--|
| $O_i$ :     |      | reallocation<br>$A \cdot s_j$ (linear)           |
| $O_{i+1}$ : |      | A (constant)                                     |
| $O_{i+2}$ : |      | A (constant)                                     |
| $O_{i+3}$ : |      | A (constant)                                     |
| $O_j$ :     |  ... | reallocation<br>$A \cdot s_j$ (earliest realloc) |

}  $\frac{s_j}{2}$  times

Table: Case 2:  $\frac{1}{2}s_j$  removes

| Array   | Costs   |
|---|---|
| $O_i$ :      | reallocation<br>$A \cdot s_j$ (linear)                |
| $O_{i+1}$ :  | $A$ (constant)  |
| $O_{i+2}$ :  | $A$ (constant)  |
| $O_{i+3}$ :  | $A$ (constant)  |
| $O_j$ :      | reallocation<br>$A \cdot s_j$ (earliest reallocation) |

$\left. \begin{array}{l} A \text{ (constant)} \\ A \text{ (constant)} \\ A \text{ (constant)} \end{array} \right\} \frac{s_j}{2} \text{ times}$

### Proof of lemma:

- If a reallocation happens at  $O_i$  and then again at  $O_j$ , then  $j - i \geq s_i/2$
- After operation  $O_i$  the capacity is

$$c_i = \left\lfloor \frac{3}{2} \cdot s_i \right\rfloor$$

- Lets consider a operation  $O_i$  to  $O_k$  with  $k - i \leq \frac{s_i}{2}$ :
  - Case 1: Since the *reallocation* we have inserted at maximum  $\text{floor}\left(\frac{1}{2} \cdot s_i\right)$  elements

$$s_k \leq s_i + \left\lfloor \frac{s_i}{2} \right\rfloor = \left\lfloor \frac{3}{2} s_i \right\rfloor = c_i \quad \text{no reallocation needed}$$

### Proof of lemma - continued:

- Case 2: Since the *reallocation* we have removed at maximum  $\left\lfloor \frac{1}{2}s_j \right\rfloor$  elements

$$s_k \geq s_j - \left\lfloor \frac{s_j}{2} \right\rfloor = \left\lceil \frac{1}{2}s_j \right\rceil$$

no reallocation needed

$$\Rightarrow 3 \cdot s_k \geq \left\lceil \frac{3}{2}s_j \right\rceil \geq \left\lfloor \frac{3}{2}s_j \right\rfloor = c_j$$

### Corollary:

$$\text{cost}(O_1) + \dots + \text{cost}(O_n) \leq 4A \cdot n$$

- Let the *reallocations* be at operations  $\text{cost}(O_{i_1}), \dots, \text{cost}(O_{i_m})$
- The **cost** of all *reallocations* are  $A \cdot (s_{i_1} + \dots + s_{i_m})$
- With the lemma we know:

$$i_2 - i_1 > \frac{s_{i_1}}{2}, \quad i_3 - i_2 > \frac{s_{i_2}}{2}, \quad \dots, \quad i_m - i_{m-1} > \frac{s_{i_{m-1}}}{2}$$

- We can conclude that:

$$i_2 - i_1 > \frac{s_{i_1}}{2} \quad \Rightarrow \quad s_{i_1} < 2(i_2 - i_1)$$

$$i_3 - i_2 > \frac{s_{i_2}}{2} \quad \Rightarrow \quad s_{i_2} < 2(i_3 - i_2)$$

$$\vdots$$

$$i_m - i_{m-1} > \frac{s_{i_{m-1}}}{2} \quad \Rightarrow \quad s_{i_{m-1}} < 2(i_m - i_{m-1})$$
$$s_{i_m} \leq n \quad (\text{trivial})$$

- The **costs** of all reallocations are:

$$\begin{aligned}\text{cost}(\text{realloc.}) &= A \cdot (s_{i_1} + \dots + s_{i_m}) \\ &< A \cdot (2(i_2 - i_1) + 2(i_3 - i_2) + \dots + 2(i_m - i_{m-1}) + n) \\ &= A \cdot (2(i_m - i_1) + n) \\ &\leq A \cdot (2n + n) = 3A \cdot n\end{aligned}$$

- Additionally we have to consider the respective constant costs for a normal append or remove ( $\leq A \cdot n$ ) therefore in total  $\leq 4 \cdot A \cdot n$

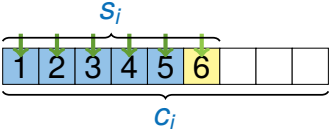
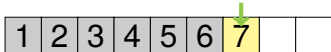
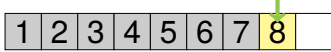
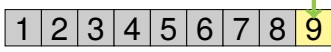

# Dynamic Arrays

## Amortized Analysis - Alternate Proof of Corollary

Table: Case 1:  $\frac{1}{2}s_j$  appends

Array

Costs

|             |   |  |                         |
|-------------|---|--|-------------------------|
| $O_i$ :     |  | reallocation<br>$A \cdot s_j$ (linear)               |                         |
| $O_{i+1}$ : |  | A (constant)   | } $\frac{s_j}{2}$ times |
| $O_{i+2}$ : |  | A (constant)   |                         |
| $O_{i+3}$ : |  | A (constant)   |                         |
| $O_j$ :     |  | reallocation<br>$A \cdot s_j$<br>(earliest realloc.) |                         |



- Total costs of  $A \cdot \frac{3}{2} \cdot s_i$  for  $\frac{s_i}{2} + 1$  operations
- Cost per operation:

$$\frac{\frac{3}{2}A \cdot s_i}{\frac{1}{2}s_i + 1} \leq \frac{\frac{3}{2}A \cdot s_i}{\frac{1}{2}s_i} = 3 \cdot A = \text{const.}$$

# Dynamic Arrays

## Amortized Analysis - Alternate Proof of Corollary



Array

Costs



- Runtime analysis for local worst-case sequence
- Same total cost as previous slide

### Bank account paradigm:

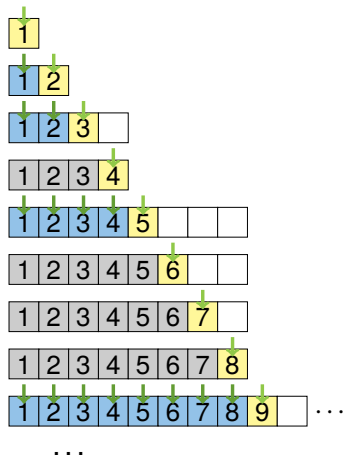
- **Idea:** “Save first, spend later”
- For each operation we deposit some coins on an “bank account”  
⇒ We still have **constant costs**
- When we have a **linear operation** (reallocation) we pay with the coins from our “bank account”
- For the “double the size” strategy we have to pay two coins per operation

# Dynamic Arrays

## Amortized Analysis - Yet Another Proof of Corollary



### Double the size:



| $\text{cost}(O_i)$ | deposit /<br>withdraw | account<br>value |
|--------------------|-----------------------|------------------|
| $O(1)$             | +2                    | 2                |
| $O(1 + 1)$         | +2 -1                 | 3                |
| $O(1 + 2)$         | +2 -2                 | 3                |
| $O(1)$             | +2                    | 5                |
| $O(1 + 4)$         | +2 -4                 | 3                |
| $O(1)$             | +2                    | 5                |
| $O(1)$             | +2                    | 7                |
| $O(1)$             | +2                    | 9                |
| $O(1 + 8)$         | +2 -8                 | 3                |
| ...                | ...                   | ...              |



Figure: Array after realloc. (insert) operation

### Why do we need to deposit 2 coins per operation?

- 1 Each newly inserted element has to be copied later (first coin)
- 2 Due to the factor of two there is for each new element also an old one in the array that also has to be copied (second coin)

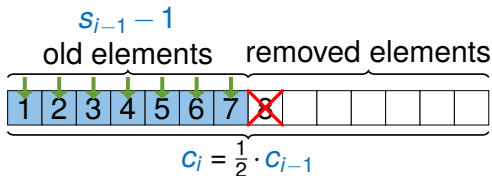


Figure: Array after realloc. (remove) operation

### Shrinking strategy: If array 1/4 full shrink by half

- How many coins do we need per *remove* operation?
- **Worst case:** The previous remove operation triggered a *reallocation*

⇒ Array is half full

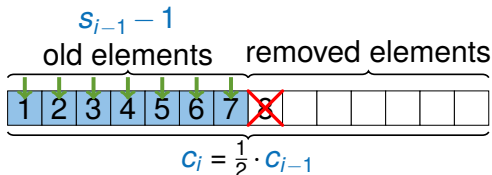


Figure: Array after realloc. (remove) operation

### Shrinking strategy: If array 1/4 full shrink by half

- Array is half full
  - The nearest *reallocation* is after removing  $\frac{1}{4}c_i$  elements
  - We have to copy  $\frac{1}{4}c_i$  elements
- ⇒ 1 coin per operation is enough

## ■ General

[CRL01] Thomas H. Cormen, Ronald L. Rivest, and Charles E. Leiserson.

**Introduction to Algorithms.**

MIT Press, Cambridge, Mass, 2001.

[MS08] Kurt Mehlhorn and Peter Sanders.

Algorithms and data structures, 2008.

<https://people.mpi-inf.mpg.de/~mehlhorn/ftp/Mehlhorn-Sanders-Toolbox.pdf>.



## ■ Amortized Analysis

[Wik] [Amortized analysis](https://en.wikipedia.org/wiki/Amortized_analysis)

https:

`//en.wikipedia.org/wiki/Amortized_analysis`