Static Arrays

Dynamic Arrays
   Introduction
   Amortized Analysis
Static Arrays

- 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>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>&quot;a&quot;</td>
<td>&quot;b&quot;</td>
<td>&quot;c&quot;</td>
<td>&quot;d&quot;</td>
<td>&quot;e&quot;</td>
</tr>
</tbody>
</table>
Python:

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

```python
# 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])
```
Static Arrays

- 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
Dynamic Arrays

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*
Dynamic Arrays
Implementation 1

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.

\[
T(n) = \sum_{i=1}^{n} T_i = \sum_{i=1}^{n} (A \cdot i) = A \cdot \sum_{i=1}^{n} i = A \cdot \frac{n^2 + n}{2} = O(n^2)
\]
Dynamic Arrays
Implementation 2

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 $C = 3$:

1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4 5 6
1 2 3 4 5 6 7

$O(1)$ 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: $C, 2 \cdot C, 3 \cdot C, \ldots$ this means:

$$T(n) = \sum_{i=1}^{n} A \cdot 1 + \sum_{i=1}^{n/C} A \cdot i \cdot C$$

$$= A \cdot n + A \cdot C \cdot \sum_{i=1}^{n/C} i$$

$$= A \cdot n + A \cdot C \cdot \frac{n^2}{2} + \frac{n}{C}$$

$$= A \cdot n + \frac{A}{2 \cdot C} \cdot n^2 + \frac{A}{2} \cdot n = O(n^2)$$

- The factor of $n^2$ is getting smaller
**Dynamic Arrays**

**Implementation 3**

**Idea:**
- Double the size of the array

```python
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?
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 $i = 0, 1, 2, \ldots, k$ with $k = \text{floor}(\log_2(n - 1))$
- In total that accounts to:

$$T(n) = n \cdot A + A \cdot \sum_{i=0}^{k} 2^i = n \cdot A + A(2^{k+1} - 1)$$

$$\leq n \cdot A + A \cdot 2^{(k+1)}$$
$$= n \cdot A + 2 \cdot A \cdot 2^{(k)}$$
$$\leq n \cdot A + 2 \cdot A \cdot n$$
$$= 3 \cdot A \cdot n$$
$$= O(n)$$
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%.
Dynamic Arrays
Shrinking

Analysis:

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

Figure: Static array with capacity $c_i$

Notation:

- We have $n$ instructions $O = \{O_1, \ldots, O_n\}$
- 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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Size $s_i$</th>
<th>Capacity $c_i$</th>
<th>Costs $\text{cost}(O_i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O_1$</td>
<td>append</td>
<td>realloc.</td>
<td>$A \cdot s_1$</td>
</tr>
<tr>
<td>$O_2$</td>
<td>append</td>
<td>$s_1$</td>
<td>$c_1$</td>
</tr>
<tr>
<td>$O_3$</td>
<td>append</td>
<td>$s_2$</td>
<td>$c_2 = c_1$</td>
</tr>
<tr>
<td>$O_4$</td>
<td>remove</td>
<td>$s_3$</td>
<td>$c_3 = c_1$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$s_4$</td>
<td>$c_4 = c_1$</td>
</tr>
<tr>
<td>$O_5$</td>
<td>remove</td>
<td>realloc.</td>
<td>$A \cdot s_5$</td>
</tr>
<tr>
<td>$O_6$</td>
<td>append</td>
<td>$s_5$</td>
<td>$c_5$</td>
</tr>
<tr>
<td>$O_7$</td>
<td>remove</td>
<td>$s_6$</td>
<td>$c_6 = c_5$</td>
</tr>
<tr>
<td>$O_8$</td>
<td>append</td>
<td>$s_7$</td>
<td>$c_7 = c_5$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$s_8$</td>
<td>$c_8 = c_5$</td>
</tr>
<tr>
<td>$O_9$</td>
<td>append</td>
<td>realloc.</td>
<td>$A \cdot s_9$</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$O_n$</td>
<td>append</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

December 2017  Prof. Dr. Rolf Backofen – Bioinformatics - University Freiburg - Germany
Dynamic Arrays
Amortized Analysis - Example

Implementation:

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

\[ s_{i-1} = 7 \]
\[ c_{i-1} = s_{i-1} = 7 \]
\[ s_i = s_{i-1} + 1 = 8 \]
\[ 12 = c_i = \left\lfloor \frac{3}{2} s_i \right\rfloor = 8 \]

Figure: Append operation with reallocation

Result: after operation we have $c_i = \frac{3}{2} \cdot s_i$
Dynamic Arrays
Amortized Analysis - Example

Implementation:

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

\[ c_{i-1} = 15 \geq 3 \cdot s_{i-1} \]
\[ s_i = s_{i-1} - 1 \]
\[ 6 = c_i = \frac{3}{2}s_i = 4 \]

**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 $2 \cdot n$
### Dynamic Arrays
Amortized Analysis - Proof

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Size</th>
<th>Capacity</th>
<th>Costs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O_1$</td>
<td>app.</td>
<td>realloc.</td>
<td>$s_1$</td>
</tr>
<tr>
<td>$O_2$</td>
<td>app.</td>
<td></td>
<td>$s_2$</td>
</tr>
<tr>
<td>$O_3$</td>
<td>app.</td>
<td></td>
<td>$s_3$</td>
</tr>
<tr>
<td>$O_4$</td>
<td>app.</td>
<td></td>
<td>$s_4$</td>
</tr>
<tr>
<td>$O_5$</td>
<td>app.</td>
<td>realloc.</td>
<td>$s_5$</td>
</tr>
<tr>
<td>$O_6$</td>
<td>app.</td>
<td></td>
<td>$s_6$</td>
</tr>
<tr>
<td>$O_7$</td>
<td>app.</td>
<td></td>
<td>$s_7$</td>
</tr>
<tr>
<td>$O_8$</td>
<td>app.</td>
<td>realloc.</td>
<td>$s_8$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

\[
4 \geq \left\lfloor \frac{s_1}{2} \right\rfloor
\]

\[
3 \geq \left\lfloor \frac{s_5}{2} \right\rfloor
\]
Dynamic Arrays
Amortized Analysis - Proof

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**: $\text{cost}(O_1) + \cdots + \text{cost}(O_n) \leq 4A \cdot n$
Dynamic Arrays
Proof: Worst Case Same Operation

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

<table>
<thead>
<tr>
<th>Array</th>
<th>Costs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O_i$:</td>
<td></td>
</tr>
<tr>
<td>$s_i$</td>
<td>reallocation</td>
</tr>
<tr>
<td>$c_i$</td>
<td>$A \cdot s_i$ (linear)</td>
</tr>
<tr>
<td>$O_{i+1}$:</td>
<td></td>
</tr>
<tr>
<td>$A$ (constant)</td>
<td></td>
</tr>
<tr>
<td>$O_{i+2}$:</td>
<td></td>
</tr>
<tr>
<td>$A$ (constant)</td>
<td></td>
</tr>
<tr>
<td>$O_{i+3}$:</td>
<td></td>
</tr>
<tr>
<td>$A$ (constant)</td>
<td></td>
</tr>
<tr>
<td>$O_j$:</td>
<td></td>
</tr>
<tr>
<td>$s_i$ times</td>
<td>reallocation $A \cdot s_j$ (earliest realloc)</td>
</tr>
</tbody>
</table>
 masslee, Dynamic Arrays, Amortized Analysis - Proof

Table: Case 2: \( \frac{1}{2}s_i \) removes

<table>
<thead>
<tr>
<th>Array</th>
<th>Costs</th>
</tr>
</thead>
<tbody>
<tr>
<td>( O_i ):</td>
<td>reallocation</td>
</tr>
<tr>
<td>( S_i )</td>
<td>( A \cdot s_i ) (linear)</td>
</tr>
<tr>
<td>( O_{i+1} ):</td>
<td>( A ) (constant)</td>
</tr>
<tr>
<td>( O_{i+2} ):</td>
<td>( A ) (constant)</td>
</tr>
<tr>
<td>( O_{i+3} ):</td>
<td>( A ) (constant)</td>
</tr>
<tr>
<td>( O_j ):</td>
<td>reallocation</td>
</tr>
<tr>
<td></td>
<td>( A \cdot s_j ) (earliest reallocation)</td>
</tr>
</tbody>
</table>

\( s_i \) \( \text{times} \)

December 2017 Prof. Dr. Rolf Backofen – Bioinformatics - University Freiburg - Germany 36 / 51
Dynamic Arrays
Amortized Analysis

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

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

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

    $$s_k \leq s_i + \left\lfloor \frac{s_i}{2} \right\rfloor = \left\lfloor \frac{3}{2} s_i \right\rfloor = c_i$$

    no reallocation needed
Proof of lemma - continued:

Case 2: Since the *reallocation* we have removed at maximum \( \left\lfloor \frac{1}{2} s_i \right\rfloor \) elements

\[
s_k \geq s_i - \left\lfloor \frac{s_i}{2} \right\rfloor = \left\lfloor \frac{1}{2} s_i \right\rfloor
\]

no reallocation needed

\[
\Rightarrow 3 \cdot s_k \geq \left\lfloor \frac{3}{2} s_i \right\rfloor \geq \left\lfloor \frac{3}{2} s_i \right\rfloor = c_i
\]
Corollary:

\[ \text{cost}(O_1) + \cdots + \text{cost}(O_n) \leq 4A \cdot n \]

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

\[
i_2 - i_1 > \frac{s_{i_1}}{2}, \quad i_3 - i_2 > \frac{s_{i_2}}{2}, \quad \ldots, \quad i_m - i_{m-1} > \frac{s_{i_{m-1}}}{2}
\]
We can conclude that:

\[ i_2 - i_1 > \frac{s_{i_1}}{2} \Rightarrow s_{i_1} < 2(i_2 - i_1) \]

\[ i_3 - i_2 > \frac{s_{i_2}}{2} \Rightarrow s_{i_2 - 2} < 2(i_3 - i_2) \]

\[ \vdots \]

\[ i_m - i_{m-1} > \frac{s_{i_{m-1}}}{2} \Rightarrow s_{i_{m-1}} < 2(i_m - i_{m-1}) \]

\[ s_{i_m} \leq n \quad \text{(trivial)} \]
The costs of all reallocations are:

\[
\text{cost}(\text{realloc.}) = A \cdot (s_1 + \cdots + s_m)
\]
\[
< A \cdot (2(i_2 - i_1) + 2(i_3 - i_2) + \cdots + 2(i_m - i_{m-1}) + n)
\]
\[
= A \cdot (2(i_m - i_1) + n)
\]
\[
\leq A \cdot (2n + n) = 3A \cdot n
\]

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: $\frac{1}{2}s_i$ appends

<table>
<thead>
<tr>
<th>Array</th>
<th>Costs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O_i$:</td>
<td>reallocation $A \cdot s_i$ (linear)</td>
</tr>
<tr>
<td>$S_i$</td>
<td></td>
</tr>
<tr>
<td>$O_{i+1}$:</td>
<td>$A$ (constant)</td>
</tr>
<tr>
<td>$C_i$</td>
<td>$s_i$ times</td>
</tr>
<tr>
<td>$O_{i+2}$:</td>
<td>$A$ (constant)</td>
</tr>
<tr>
<td>$O_{i+3}$:</td>
<td>$A$ (constant)</td>
</tr>
<tr>
<td>$O_j$:</td>
<td>reallocation $A \cdot s_j$</td>
</tr>
</tbody>
</table>

(earliest realloc.)
Total costs of $A \cdot \frac{3}{2} \cdot s_i$ for $\frac{s_i}{2} + 1$ operations

Cost per operation:

\[
\frac{\frac{3}{2}A \cdot s_i}{\frac{1}{2}s_i + 1} < \frac{\frac{3}{2}A \cdot s_i}{\frac{1}{2}s_i} = 3 \cdot A = \text{const.}
\]
### Dynamic Arrays

**Amortized Analysis - Alternate Proof of Corollary**

#### Array Costs

<table>
<thead>
<tr>
<th>$O_i$</th>
<th>$O_{i+1}$</th>
<th>$O_{i+2}$</th>
<th>$O_{i+3}$</th>
<th>$O_j$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6</td>
<td>1 2 3 4 5 6</td>
<td>1 2 3 4 5</td>
<td>1 2 3 4</td>
<td>1 2 3</td>
</tr>
</tbody>
</table>

- $s_i$: reallocation
- $A \cdot s_i$ (linear)

- $c_i$: $A$ (constant)

- $s_i$ times

#### Runtime Analysis

- 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”
  \[ \Rightarrow \text{We still have constant costs} \]
- 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:

<table>
<thead>
<tr>
<th>cost($O_i$)</th>
<th>deposit / withdraw</th>
<th>account value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>+2</td>
<td>2</td>
</tr>
<tr>
<td>$O(1 + 1)$</td>
<td>+2 -1</td>
<td>3</td>
</tr>
<tr>
<td>$O(1 + 2)$</td>
<td>+2 -2</td>
<td>3</td>
</tr>
<tr>
<td>$O(1)$</td>
<td>+2</td>
<td>5</td>
</tr>
<tr>
<td>$O(1 + 4)$</td>
<td>+2 -4</td>
<td>3</td>
</tr>
<tr>
<td>$O(1)$</td>
<td>+2</td>
<td>5</td>
</tr>
<tr>
<td>$O(1)$</td>
<td>+2</td>
<td>7</td>
</tr>
<tr>
<td>$O(1)$</td>
<td>+2</td>
<td>9</td>
</tr>
<tr>
<td>$O(1 + 8)$</td>
<td>+2 -8</td>
<td>3</td>
</tr>
</tbody>
</table>

...
Dynamic Arrays
Amortized Analysis - Yet Another Proof of Corollary

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. (insert) operation
Dynamic Arrays

Amortized Analysis - Yet Another Proof of Corollary

\[ C_i = \frac{1}{2} \cdot C_{i-1} \]

\[ s_{i-1} - 1 \]
old elements

\[ \text{removed elements} \]

\[ 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad \times \]

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*

\[ \Rightarrow \text{Array is half full} \]
Dynamic Arrays
Amortized Analysis - Yet Another Proof of Corollary

\[ c_i = \frac{1}{2} \cdot c_{i-1} \]

\[ s_{i-1} - 1 \]

old elements

removed elements

\[
\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & \times \\
\end{array}
\]

\[ C_i = \frac{1}{2} \cdot C_{i-1} \]

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

\[ \Rightarrow 1 \text{ coin per operation is enough} \]
Further Literature

General


Further Literature

- Amortized Analysis
  
  [Wik] Amortized analysis
  https://en.wikipedia.org/wiki/Amortized_analysis