Amortized Analysis

EECS 214

November 11–13, 2015

Take-aways

- What is *amortized time*?
- How does amortized time differ from *average time*?
- When is amortized time useful, and when might we want to avoid it?
- How can we figure out the amortized time of data structure operations?
- How does a dynamic array achieve its amortized time complexity?

Example: dynamic arrays

Language	**Type**
C++	`std::vector`
Java	`ArrayList`
Python	`list`
Ruby	`Array`

Example: dynamic arrays

Language	**Type**
C++	`std::vector`
Java	`ArrayList`
Python	`list`
Ruby	`Array`
C	*you're on your own*

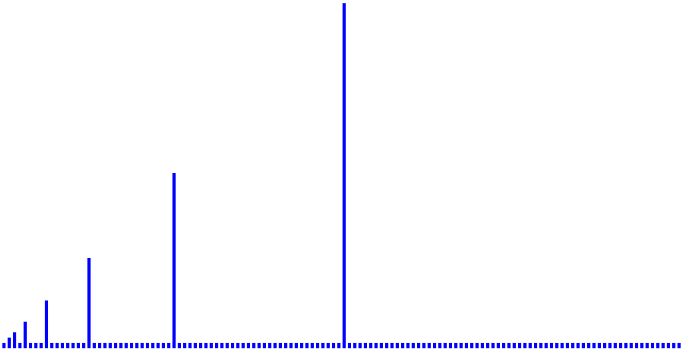
Example: dynamic arrays

Language	**Type**
C++	`std::vector`
Java	`ArrayList`
Python	`list`
Ruby	`Array`
C	*you’re on your own*
ASL	*you’re on your own*

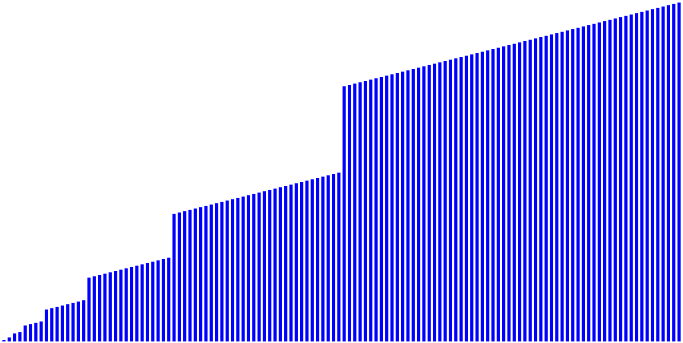
Iteratively growing a dynamic array

```
std::vector<int> v;
for (int i = 0; i < N; ++i) v.push_back(i);

ArrayList<Integer> v = new ArrayList<>();
for (int i = 0; i < N; ++i) v.add(i);

v = list()
for i in range(0, 10): v.append(i)

v = Array.new
for i in 0 ... N do v.push(i) end
```

Time per operation



Accumulated time



What's it doing?

- A dynamic array is backed by a fixed-size array with excess capacity:
  `(define-struct dynarray [data size])`
- When the array fills, allocate a fixed-size array that's twice as big and copy over the elements.

Time complexity of a single insertion

A single insertion:

$$T_{\text{insert}}(n) = \mathcal{O}(n)$$

Time complexity of a sequence of insertions

Hence, for a sequence of insertions:

$$T_{\text{insert}-\text{sequence}}(m) = \sum_{i=1}^{m} \mathcal{O}(i)$$

Time complexity of a sequence of insertions

Hence, for a sequence of insertions:

$$
\begin{aligned}
T_{\text{insert}-\text{sequence}}(m) &= \sum_{i=1}^{m} \mathcal{O}(i) \\
&= \mathcal{O}\left(\sum_{i=1}^{m} i\right)
\end{aligned}
$$

Time complexity of a sequence of insertions

Hence, for a sequence of insertions:

$$\begin{aligned}
T_{\text{insert}-\text{sequence}}(m) &= \sum_{i=1}^{m} \mathcal{O}(i) \\
&= \mathcal{O}\left(\sum_{i=1}^{m} i\right) \\
&= \mathcal{O}(1 + 2 + \cdots + (m-1) + m)
\end{aligned}$$

Time complexity of a sequence of insertions

Hence, for a sequence of insertions:

$$\begin{aligned} T_{\text{insert}-\text{sequence}}(m) &= \sum_{i=1}^{m} \mathcal{O}(i) \\ &= \mathcal{O}\left(\sum_{i=1}^{m} i\right) \\ &= \mathcal{O}(1 + 2 + \cdots + (m-1) + m) \\ &= \mathcal{O}\left(\frac{m(m+1)}{2}\right) \end{aligned}$$

Time complexity of a sequence of insertions

Hence, for a sequence of insertions:

$$
\begin{aligned}
T_{\text{insert}-\text{sequence}}(m) &= \sum_{i=1}^{m} \mathcal{O}(i) \\
&= \mathcal{O}\left(\sum_{i=1}^{m} i\right) \\
&= \mathcal{O}(1 + 2 + \cdots + (m-1) + m) \\
&= \mathcal{O}\left(\frac{m(m+1)}{2}\right) \\
&= \mathcal{O}(m^2)
\end{aligned}
$$

Amortized time complexity

Amortized time complexity considers the cost of a sequence of operations by paying attention to the state of the data structure.

Amortized time complexity

Amortized time complexity considers the cost of a sequence of operations by paying attention to the state of the data structure.

Then it apportions the time evenly among the operations.

Amortized time complexity

Amortized time complexity considers the cost of a sequence of operations by paying attention to the state of the data structure.

Then it apportions the time evenly among the operations.

Amortization is about the *worst case*, not merely the *average case*.

Banker's method: real costs vs. accounting costs

Let c_i be the actual cost of the ith operation
Let c'_i be the charged cost of the ith operation

Banker's method: real costs vs. accounting costs

Let c_i be the actual cost of the ith operation

Let c'_i be the charged cost of the ith operation—we choose this!

Banker's method: real costs vs. accounting costs

Let c_i be the actual cost of the ith operation
Let c_i' be the charged cost of the ith operation—we choose this!

If total actual cost does not exceed the total charged cost,

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} c_i' ,$$

then we say that the ith operation has worst-case *amortized* time $\mathcal{O}(c_i')$,

Amortized time for dynamic array insertion (banker style)

Consider the ith insert operation (which results in size i):

i	1	2	3	4	5	6	7	8	9	10

Amortized time for dynamic array insertion (banker style)

Consider the ith insert operation (which results in size i):

i	1	2	3	4	5	6	7	8	9	10
cap_i	1	2	4	4	8	8	8	8	16	16

Let cap_i be the capacity after operation i

Amortized time for dynamic array insertion (banker style)

Consider the ith insert operation (which results in size i):

i	1	2	3	4	5	6	7	8	9	10
cap_i	1	2	4	4	8	8	8	8	16	16
c_i	1	2	3	1	5	1	1	1	9	1

Let cap_i be the capacity after operation i

Let c_i be the actual cost of the ith operation (number of elements inserted or copied)

Amortized time for dynamic array insertion (banker style)

Consider the ith insert operation (which results in size i):

i	1	2	3	4	5	6	7	8	9	10
cap_i	1	2	4	4	8	8	8	8	16	16
c_i	1	2	3	1	5	1	1	1	9	1
c'_i	1	1	1	1	1	1	1	1	1	1

Let cap_i be the capacity after operation i

Let c_i be the actual cost of the ith operation (number of elements inserted or copied)

Let c'_i be the charged cost of the ith operation—we choose a constant to cover the cost of large future operations

Amortized time for dynamic array insertion (banker style)

Consider the ith insert operation (which results in size i):

i	1	2	3	4	5	6	7	8	9	10
cap_i	1	2	4	4	8	8	8	8	16	16
c_i	1	2	3	1	5	1	1	1	9	1
${c'}_i$	1	1	1	1	1	1	1	1	1	1
bal_i	0	-1	-1	-1	-1	-1	-1	...		

Let cap_i be the capacity after operation i

Let c_i be the actual cost of the ith operation (number of elements inserted or copied)

Let c'_i be the charged cost of the ith operation—we choose a constant to cover the cost of large future operations

Let bal_i be the balance: $bal_i = bal_{i-1} - c_i + c'_i$.

Amortized time for dynamic array insertion (banker style)

Consider the ith insert operation (which results in size i):

i	1	2	3	4	5	6	7	8	9	10
cap_i	1	2	4	4	8	8	8	8	16	16
c_i	1	2	3	1	5	1	1	1	9	1
c'_i	2	2	2	2	2	2	2	2	2	2
bal_i	1	1	0	1	-1	-1	0	1	-1	-1

Let cap_i be the capacity after operation i

Let c_i be the actual cost of the ith operation (number of elements inserted or copied)

Let c_i' be the charged cost of the ith operation—we choose a constant to cover the cost of large future operations

Let bal_i be the balance: $bal_i = bal_{i-1} - c_i + c_i'$.

Amortized time for dynamic array insertion (banker style)

Consider the ith insert operation (which results in size i):

i	1	2	3	4	5	6	7	8	9	10
cap_i	1	2	4	4	8	8	8	8	16	16
c_i	1	2	3	1	5	1	1	1	9	1
c'_i	3	3	3	3	3	3	3	3	3	3
bal_i	2	3	3	5	3	5	7	9	3	5

Let cap_i be the capacity after operation i

Let c_i be the actual cost of the ith operation (number of elements inserted or copied)

Let c'_i be the charged cost of the ith operation—we choose a constant to cover the cost of large future operations

Let bal_i be the balance: $bal_i = bal_{i-1} - c_i + c'_i$.

Amortized time for dynamic array insertion (banker style)

Consider the ith insert operation (which results in size i):

i	1	2	3	4	5	6	7	8	9	10
cap_i	1	2	4	4	8	8	8	8	16	16
c_i	2	4	7	1	13	1	1	1	25	1
c'_i	3	3	3	3	3	3	3	3	3	3
bal_i	1	0	-1	-1	$\ldots$					

Let cap_i be the capacity after operation i

Let c_i be the actual cost of the ith operation (number of elements inserted or copied)

Let c'_i be the charged cost of the ith operation—we choose a constant to cover the cost of large future operations

Let bal_i be the balance: $bal_i = bal_{i-1} - c_i + c'_i$.

Amortized time for dynamic array insertion (banker style)

Consider the ith insert operation (which results in size i):

i	1	2	3	4	5	6	7	8	9	10
cap_i	1	2	4	4	8	8	8	8	16	16
c_i	2	4	7	1	13	1	1	1	25	1
c'_i	5	5	5	5	5	5	5	5	5	5
bal_i	3	4	2	6	-2	$\dots$				

Let cap_i be the capacity after operation i

Let c_i be the actual cost of the ith operation (number of elements inserted or copied)

Let c'_i be the charged cost of the ith operation—we choose a constant to cover the cost of large future operations

Let bal_i be the balance: $bal_i = bal_{i-1} - c_i + c'_i$.

Amortized time for dynamic array insertion (banker style)

Consider the ith insert operation (which results in size i):

i	1	2	3	4	5	6	7	8	9	10
cap_i	1	2	4	4	8	8	8	8	16	16
c_i	2	4	7	1	13	1	1	1	25	1
c'_i	7	7	7	7	7	7	7	7	7	7
bal_i	5	8	8	14	8	14	20	26	8	14

Let cap_i be the capacity after operation i

Let c_i be the actual cost of the ith operation (number of elements inserted or copied)

Let c'_i be the charged cost of the ith operation—we choose a constant to cover the cost of large future operations

Let bal_i be the balance: $bal_i = bal_{i-1} - c_i + c'_i$.

Physicist's method: potential "energy"

We define a potential function Φ on data structure states, where:

$$\Phi(v_0) = 0 \qquad \text{starts at } 0$$

$$\Phi(v_t) \geq 0 \qquad \text{never goes negative}$$

Physicist's method: potential "energy"

We define a potential function Φ on data structure states, where:

$$\begin{aligned} \Phi(v_0) &= 0 && \text{starts at 0} \\ \Phi(v_t) &\geq 0 && \text{never goes negative} \end{aligned}$$

Φ is akin to the balance in the banker's method, but history-less

Physicist's method: potential "energy"

We define a potential function Φ on data structure states, where:

$$\begin{aligned} \Phi(v_0) &= 0 && \text{starts at 0} \\ \Phi(v_t) &\geq 0 && \text{never goes negative} \end{aligned}$$

Φ is akin to the balance in the banker's method, but history-less

We then define the amortized time of an operation:

$$\begin{aligned} c'_i &= c_i + \Phi(v_i) - \Phi(v_{i-1}) \\ &= c_i + \Delta\Phi(v_i) \end{aligned}$$

Physicist's method: potential "energy"

We define a potential function Φ on data structure states, where:

$$\begin{aligned} \Phi(v_0) &= 0 && \text{starts at 0} \\ \Phi(v_t) &\geq 0 && \text{never goes negative} \end{aligned}$$

Φ is akin to the balance in the banker's method, but history-less

We then define the amortized time of an operation:

$$\begin{aligned} c'_i &= c_i + \Phi(v_i) - \Phi(v_{i-1}) \\ &= c_i + \Delta\Phi(v_i) \end{aligned}$$

Potential function for dynamic arrays

We choose a potential function

$$\Phi(v) = 2n - m\ ,$$

where n is the size and m the capacity of v.

Potential function for dynamic arrays

We choose a potential function

$$\Phi(v) = 2n - m ,$$

where n is the size and m the capacity of v.

Let's check Φ's properties:

Potential function for dynamic arrays

We choose a potential function

$$\Phi(v) = 2n - m ,$$

where n is the size and m the capacity of v.

Let's check Φ's properties:

- ✓ The initial vector has no size and no capacity, so $\Phi(v_0) = 0$

Potential function for dynamic arrays

We choose a potential function

$$\Phi(v) = 2n - m \,,$$

where n is the size and m the capacity of v.

Let's check Φ's properties:

- ✓ The initial vector has no size and no capacity, so $\Phi(v_0) = 0$

  The capacity is never more than twice the size, because we double when it's full

Potential function for dynamic arrays

We choose a potential function

$$\Phi(v) = 2n - m\,,$$

where n is the size and m the capacity of v.

Let's check Φ's properties:

- ✓ The initial vector has no size and no capacity, so $\Phi(v_0) = 0$

  The capacity is never more than twice the size, because we double when it's full; hence $2n \geq m$

Potential function for dynamic arrays

We choose a potential function

$$\Phi(v) = 2n - m\,,$$

where n is the size and m the capacity of v.

Let's check Φ's properties:

- ✓ The initial vector has no size and no capacity, so $\Phi(v_0) = 0$
- ✓ The capacity is never more than twice the size, because we double when it's full; hence $2n \geq m$; hence $\Phi(v) = 2n - m \geq 0$.

Amortized time for dynamic array insertion (physicist style)

Let's compute c_i' for insertion. Remember that $c_i' = c_i + \Phi(v_i) - \Phi(v_{i-1})$. There are two possibilities:

Amortized time for dynamic array insertion (physicist style)

Let's compute c_i' for insertion. Remember that $c_i' = c_i + \Phi(v_i) - \Phi(v_{i-1})$. There are two possibilities:

If $n < m$ then $c_i = 1$.

Amortized time for dynamic array insertion (physicist style)

Let's compute c_i' for insertion. Remember that $c_i' = c_i + \Phi(v_i) - \Phi(v_{i-1})$. There are two possibilities:

If $n < m$ then $c_i = 1$. So

$$c_i' = 1 + (2(n+1) - m) - (2n - m)$$

Amortized time for dynamic array insertion (physicist style)

Let's compute c_i' for insertion. Remember that $c_i' = c_i + \Phi(v_i) - \Phi(v_{i-1})$. There are two possibilities:

- ✓ If $n < m$ then $c_i = 1$. So

$$\begin{aligned} c_i' &= 1 + (2(n+1) - m) - (2n - m) \\ &= 1 + 2 = 3 \end{aligned}$$

Amortized time for dynamic array insertion (physicist style)

Let's compute c_i' for insertion. Remember that $c_i' = c_i + \Phi(v_i) - \Phi(v_{i-1})$. There are two possibilities:

✓ If $n < m$ then $c_i = 1$. So

$$\begin{aligned} c_i' &= 1 + (2(n+1) - m) - (2n - m) \\ &= 1 + 2 = 3 \end{aligned}$$

If $n = m$ then $c_i = n + 1$ (copy plus simple insert).

Amortized time for dynamic array insertion (physicist style)

Let's compute c_i' for insertion. Remember that $c_i' = c_i + \Phi(v_i) - \Phi(v_{i-1})$. There are two possibilities:

✓ If $n < m$ then $c_i = 1$. So

$$\begin{aligned} c_i' &= 1 + (2(n+1) - m) - (2n - m) \\ &= 1 + 2 = 3 \end{aligned}$$

If $n = m$ then $c_i = n + 1$ (copy plus simple insert). So

$$c_i' = n + 1 + (2(n+1) - 2m) - (2n - m)$$

Amortized time for dynamic array insertion (physicist style)

Let's compute c_i' for insertion. Remember that $c_i' = c_i + \Phi(v_i) - \Phi(v_{i-1})$. There are two possibilities:

✓ If $n < m$ then $c_i = 1$. So

$$\begin{aligned} c_i' &= 1 + (2(n+1) - m) - (2n - m) \\ &= 1 + 2 = 3 \end{aligned}$$

If $n = m$ then $c_i = n + 1$ (copy plus simple insert). So

$$\begin{aligned} c_i' &= n + 1 + (2(n+1) - 2m) - (2n - m) \\ &= n + 1 + (2(n+1) - 2n) - (2n - n) \qquad \text{because } n = m \end{aligned}$$

Amortized time for dynamic array insertion (physicist style)

Let's compute c'_i for insertion. Remember that $c'_i = c_i + \Phi(v_i) - \Phi(v_{i-1})$. There are two possibilities:

- ✓ If $n < m$ then $c_i = 1$. So

$$\begin{aligned} c'_i &= 1 + (2(n+1) - m) - (2n - m) \\ &= 1 + 2 = 3 \end{aligned}$$

- ✓ If $n = m$ then $c_i = n + 1$ (copy plus simple insert). So

$$\begin{aligned} c'_i &= n + 1 + (2(n+1) - 2m) - (2n - m) \\ &= n + 1 + (2(n+1) - 2n) - (2n - n) \qquad \text{because } n = m \\ &= 1 + 2 + n + 2n - 2n + 2n - n = 3 \end{aligned}$$

Another example: (naïve) persistent banker's queue

A data structure is *persistent* when modifications do not destroy the previous state of the structure.

Another example: (naïve) persistent banker's queue

A data structure is *persistent* when modifications do not destroy the previous state of the structure.(The opposite is *ephemeral*.)

Another example: (naïve) persistent banker's queue

A data structure is *persistent* when modifications do not destroy the previous state of the structure.(The opposite is *ephemeral*.)

What if we want a persistent FIFO queue with sub-linear operations?

Take-aways

- What is *amortized time*?
- How does amortized time differ from *average time*?
- When is amortized time useful, and when might we want to avoid it?
- How can we figure out the amortized time of data structure operations?
- How does a dynamic array achieve its amortized time complexity?