

# Basic Data Structures: Dynamic Arrays and Amortized Analysis

Neil Rhodes

Department of Computer Science and Engineering  
University of California, San Diego

Data Structures  
Data Structures and Algorithms

# Outline

- 1 Dynamic Arrays
- 2 Amortized Analysis—Aggregate Method
- 3 Amortized Analysis—Banker's Method
- 4 Amortized Analysis—Physicist's Method

Problem: static arrays are static!

```
int my_array[100];
```

Problem: static arrays are static!

```
int my_array[100];
```

Semi-solution: dynamically-allocated arrays:

```
int *my_array = new int[size];
```

Problem: might not know max size when allocating an array

Problem: might not know max size when allocating an array

*All problems in computer science  
can be solved by another level of  
indirection.*

Problem: might not know max size when allocating an array

*All problems in computer science can be solved by another level of indirection.*

Solution: *dynamic arrays* (also known as *resizable arrays*)

Idea: store a pointer to a dynamically allocated array, and replace it with a newly-allocated array as needed.

# Definition

## Dynamic Array:

Abstract data type with the following operations (at a minimum):

---

\*must be constant time



# Definition

## Dynamic Array:

Abstract data type with the following operations (at a minimum):

- $\text{Get}(i)$ : returns element at location  $i^*$

---

\*must be constant time

# Definition

## Dynamic Array:

Abstract data type with the following operations (at a minimum):

- $\text{Get}(i)$ : returns element at location  $i^*$
- $\text{Set}(i, val)$ : Sets element  $i$  to  $val^*$

---

\*must be constant time

# Definition

## Dynamic Array:

Abstract data type with the following operations (at a minimum):

- $\text{Get}(i)$ : returns element at location  $i^*$
- $\text{Set}(i, val)$ : Sets element  $i$  to  $val^*$
- $\text{PushBack}(val)$ : Adds  $val$  to the end

---

\*must be constant time

# Definition

## Dynamic Array:

Abstract data type with the following operations (at a minimum):

- $\text{Get}(i)$ : returns element at location  $i^*$
- $\text{Set}(i, val)$ : Sets element  $i$  to  $val^*$
- $\text{PushBack}(val)$ : Adds  $val$  to the end
- $\text{Remove}(i)$ : Removes element at location  $i$

---

\*must be constant time

# Definition

## Dynamic Array:

Abstract data type with the following operations (at a minimum):

- $\text{Get}(i)$ : returns element at location  $i^*$
- $\text{Set}(i, val)$ : Sets element  $i$  to  $val^*$
- $\text{PushBack}(val)$ : Adds  $val$  to the end
- $\text{Remove}(i)$ : Removes element at location  $i$
- $\text{Size}()$ : the number of elements

---

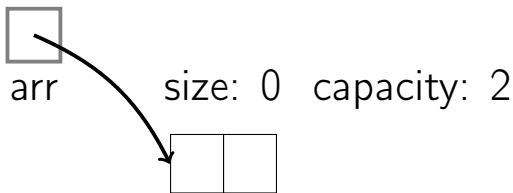
\*must be constant time

# Implementation

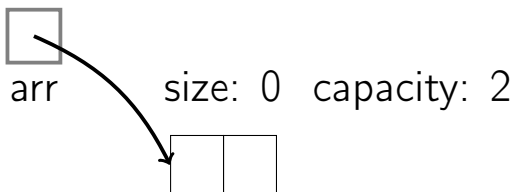
Store:

- `arr`: dynamically-allocated array
- `capacity`: size of the dynamically-allocated array
- `size`: number of elements currently in the array

# Dynamic Array Resizing



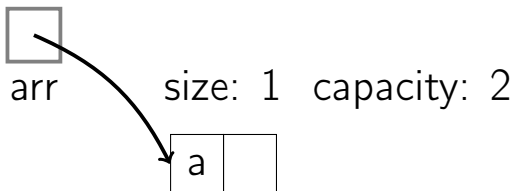
# Dynamic Array Resizing



PushBack(a)

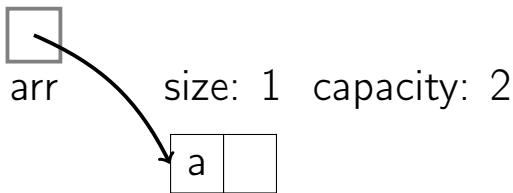


# Dynamic Array Resizing

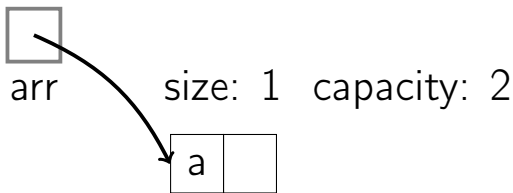


`PushBack(a)`

# Dynamic Array Resizing

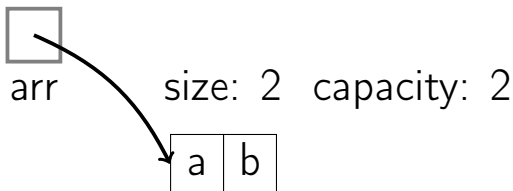


# Dynamic Array Resizing



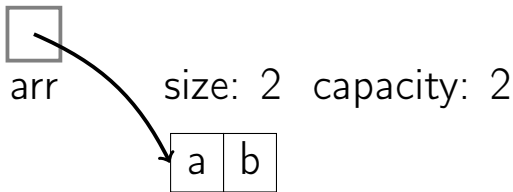
PushBack(b)

# Dynamic Array Resizing

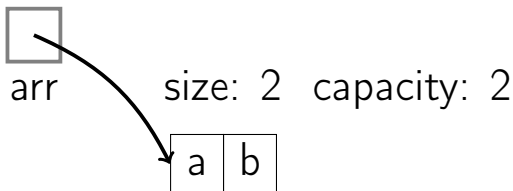


`PushBack(b)`

# Dynamic Array Resizing

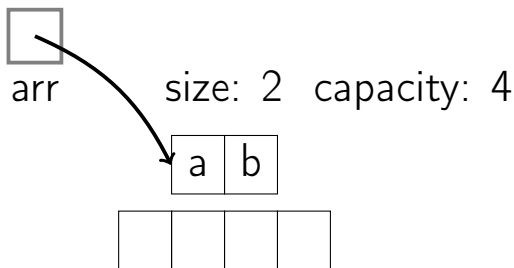


# Dynamic Array Resizing



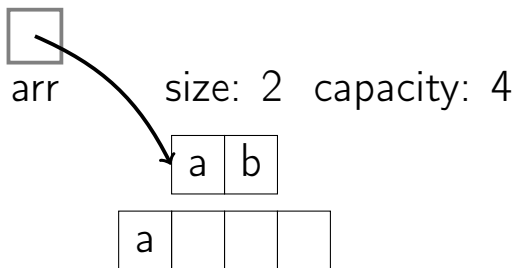
PushBack(c)

# Dynamic Array Resizing



PushBack(c)

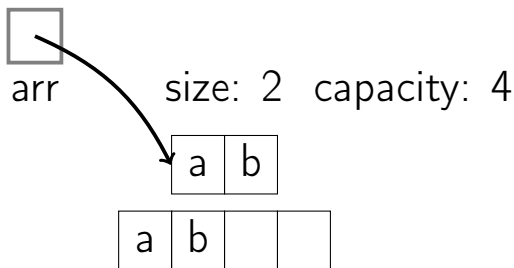
# Dynamic Array Resizing



PushBack(c)

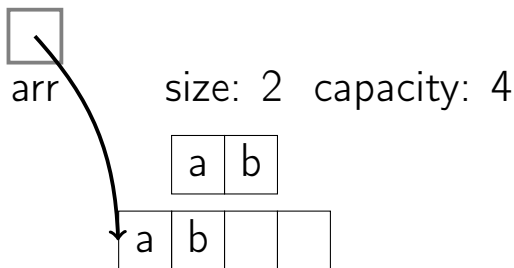


# Dynamic Array Resizing



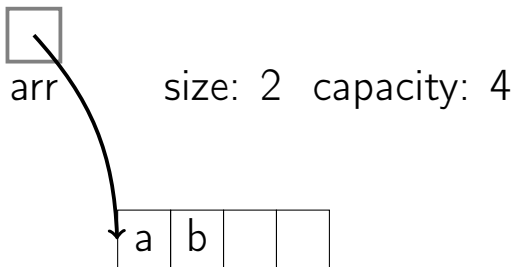
`PushBack(c)`

# Dynamic Array Resizing



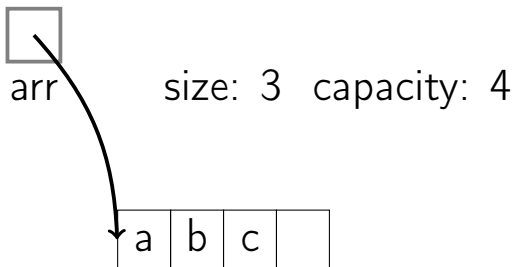
PushBack(c)

# Dynamic Array Resizing



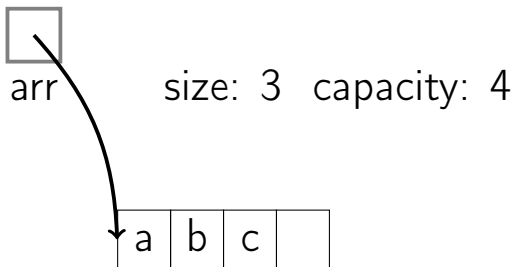
`PushBack(c)`

# Dynamic Array Resizing

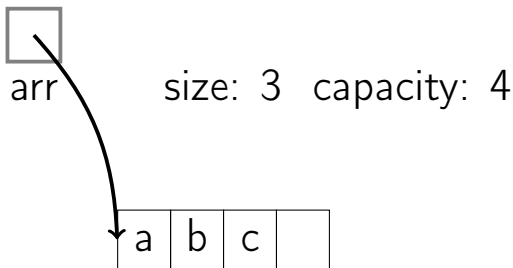


`PushBack(c)`

# Dynamic Array Resizing

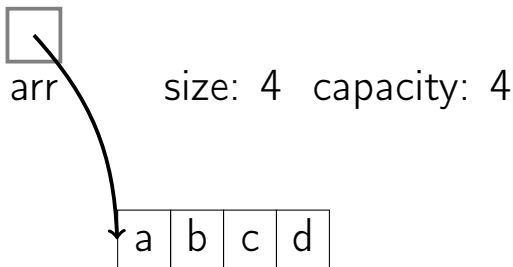


# Dynamic Array Resizing



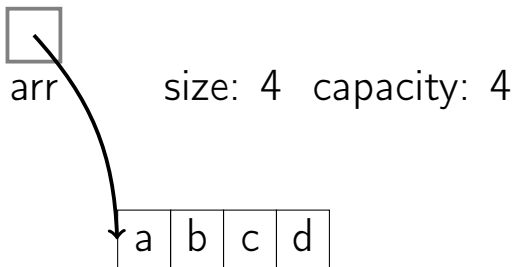
PushBack(d)

# Dynamic Array Resizing



`PushBack(d)`

# Dynamic Array Resizing



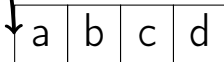


# Dynamic Array Resizing



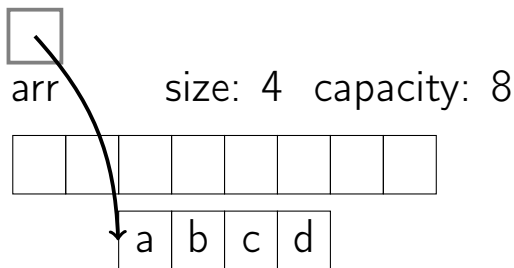
arr

size: 4 capacity: 4



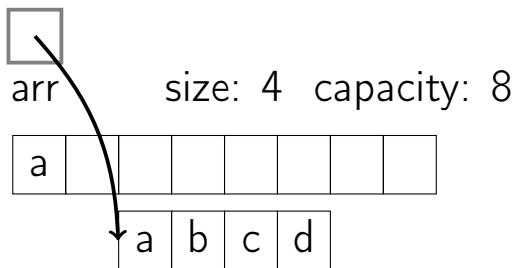
PushBack(e)

# Dynamic Array Resizing



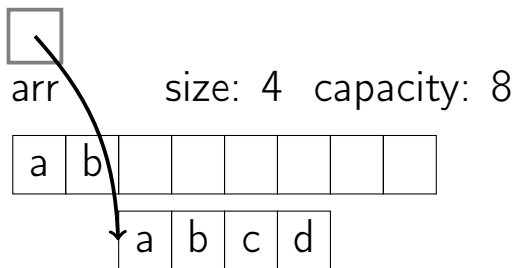
PushBack(e)

# Dynamic Array Resizing



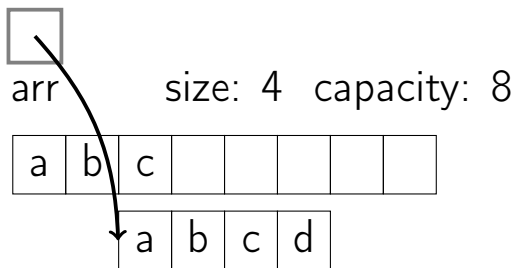
PushBack(e)

# Dynamic Array Resizing



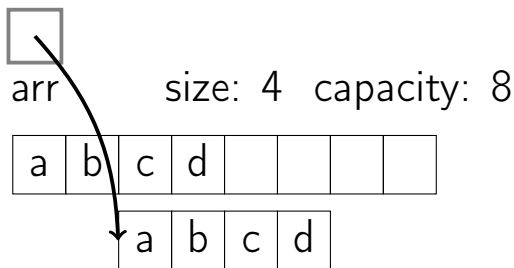
PushBack(e)

# Dynamic Array Resizing



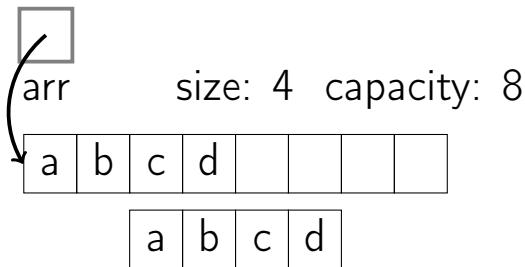
PushBack(e)

# Dynamic Array Resizing



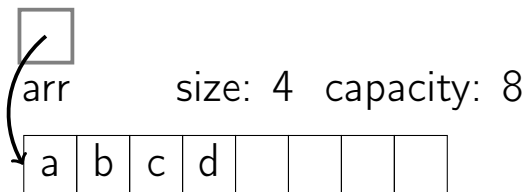
PushBack(e)

# Dynamic Array Resizing



PushBack(e)

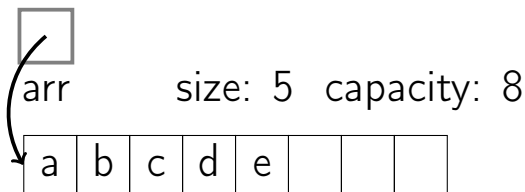
# Dynamic Array Resizing



PushBack(e)



# Dynamic Array Resizing



PushBack(e)

Get(*i*)

```
if  $i < 0$  or  $i \geq size$ :
```

```
    ERROR: index out of range
```

```
return arr[i]
```

*Set( $i$ ,  $val$ )*

if  $i < 0$  or  $i \geq size$ :

    ERROR: index out of range

$arr[i] = val$

## PushBack(*val*)

```
if size = capacity:  
    allocate new_arr[ $2 \times \textit{capacity}$ ]  
    for i from 0 to size - 1:  
        new_arr[i]  $\leftarrow$  arr[i]  
    free arr  
    arr  $\leftarrow$  new_arr; capacity  $\leftarrow 2 \times \textit{capacity}$   
arr[size]  $\leftarrow$  val  
size  $\leftarrow$  size + 1
```

## Remove(*i*)

if  $i < 0$  or  $i \geq \text{size}$ :

    ERROR: index out of range

for  $j$  from  $i$  to  $\text{size} - 2$ :

$\text{arr}[j] \leftarrow \text{arr}[j + 1]$

$\text{size} \leftarrow \text{size} - 1$

```
Size()
```

```
return size
```

# Common Implementations

- C++: `vector`
- Java: `ArrayList`
- Python: `list` (the only kind of array)

# Runtimes

Get( $i$ ) |  $O(1)$



# Runtimes

Get( <i>i</i> )	$O(1)$
Set( <i>i</i> , <i>val</i> )	$O(1)$

# Runtimes

Get( $i$ )	$O(1)$
Set( $i, val$ )	$O(1)$
PushBack( $val$ )	$O(n)$

# Runtimes

Get( $i$ )	$O(1)$
Set( $i, val$ )	$O(1)$
PushBack( $val$ )	$O(n)$
Remove( $i$ )	$O(n)$

# Runtimes

Get( $i$ )	$O(1)$
Set( $i$ , $val$ )	$O(1)$
PushBack( $val$ )	$O(n)$
Remove( $i$ )	$O(n)$
Size()	$O(1)$

# Summary

- Unlike static arrays, dynamic arrays can be resized.

# Summary

- Unlike static arrays, dynamic arrays can be resized.
- Appending a new element to a dynamic array is often constant time, but can take  $O(n)$ .

# Summary

- Unlike static arrays, dynamic arrays can be resized.
- Appending a new element to a dynamic array is often constant time, but can take  $O(n)$ .
- Some space is wasted

# Summary

- Unlike static arrays, dynamic arrays can be resized.
- Appending a new element to a dynamic array is often constant time, but can take  $O(n)$ .
- Some space is wasted



# Summary

- Unlike static arrays, dynamic arrays can be resized.
- Appending a new element to a dynamic array is often constant time, but can take  $O(n)$ .
- Some space is wasted—at most half.

# Outline

- ① Dynamic Arrays
- ② Amortized Analysis—Aggregate Method
- ③ Amortized Analysis—Banker's Method
- ④ Amortized Analysis—Physicist's Method

Sometimes, looking at the individual worst-case may be too severe. We may want to know the total worst-case cost for a sequence of operations.

## Dynamic Array

We only resize every so often.

Many  $O(1)$  operations are followed by an  $O(n)$  operations.

What is the total cost of inserting many elements?

## Definition

Amortized cost: Given a sequence of  $n$  operations, the amortized cost is:

$$\frac{\text{Cost}(n \text{ operations})}{n}$$

# Aggregate Method

Dynamic array:  $n$  calls to PushBack

# Aggregate Method

Dynamic array:  $n$  calls to PushBack

Let  $c_i =$  cost of  $i$ 'th insertion.

# Aggregate Method

Dynamic array:  $n$  calls to PushBack

Let  $c_i =$  cost of  $i$ 'th insertion.

$$c_i = 1 + \left\{ \right.$$



# Aggregate Method

Dynamic array:  $n$  calls to PushBack

Let  $c_i =$  cost of  $i$ 'th insertion.

$$c_i = 1 + \begin{cases} i - 1 & \text{if } i - 1 \text{ is a power of 2} \end{cases}$$

# Aggregate Method

Dynamic array:  $n$  calls to PushBack

Let  $c_i =$  cost of  $i$ 'th insertion.

$$c_i = 1 + \begin{cases} i - 1 & \text{if } i - 1 \text{ is a power of 2} \\ 0 & \text{otherwise} \end{cases}$$

# Aggregate Method

Dynamic array:  $n$  calls to PushBack

Let  $c_i =$  cost of  $i$ 'th insertion.

$$c_i = 1 + \begin{cases} i - 1 & \text{if } i - 1 \text{ is a power of 2} \\ 0 & \text{otherwise} \end{cases}$$

$$\frac{\sum_{i=1}^n c_i}{n}$$

# Aggregate Method

Dynamic array:  $n$  calls to PushBack

Let  $c_i = \text{cost of } i\text{'th insertion.}$

$$c_i = 1 + \begin{cases} i - 1 & \text{if } i - 1 \text{ is a power of 2} \\ 0 & \text{otherwise} \end{cases}$$

$$\frac{\sum_{i=1}^n c_i}{n} = \frac{n + \sum_{j=1}^{\lfloor \log_2(n-1) \rfloor} 2^j}{n}$$

# Aggregate Method

Dynamic array:  $n$  calls to PushBack

Let  $c_i = \text{cost of } i\text{'th insertion.}$

$$c_i = 1 + \begin{cases} i - 1 & \text{if } i - 1 \text{ is a power of 2} \\ 0 & \text{otherwise} \end{cases}$$

$$\frac{\sum_{i=1}^n c_i}{n} = \frac{n + \sum_{j=1}^{\lfloor \log_2(n-1) \rfloor} 2^j}{n} = \frac{O(n)}{n}$$

# Aggregate Method

Dynamic array:  $n$  calls to PushBack

Let  $c_i = \text{cost of } i\text{'th insertion.}$

$$c_i = 1 + \begin{cases} i - 1 & \text{if } i - 1 \text{ is a power of 2} \\ 0 & \text{otherwise} \end{cases}$$

$$\frac{\sum_{i=1}^n c_i}{n} = \frac{n + \sum_{j=1}^{\lfloor \log_2(n-1) \rfloor} 2^j}{n} = \frac{O(n)}{n} = O(1)$$

# Outline

- 1 Dynamic Arrays
- 2 Amortized Analysis—Aggregate Method
- 3 Amortized Analysis—Banker's Method
- 4 Amortized Analysis—Physicist's Method

# Banker's Method

- Charge extra for each cheap operation.



# Banker's Method

- Charge extra for each cheap operation.
- Save the extra charge as tokens in your data structure (conceptually).

# Banker's Method

- Charge extra for each cheap operation.
- Save the extra charge as tokens in your data structure (conceptually).
- Use the tokens to pay for expensive operations.

# Banker's Method

- Charge extra for each cheap operation.
- Save the extra charge as tokens in your data structure (conceptually).
- Use the tokens to pay for expensive operations.

Like an amortizing loan.

# Banker's Method

Dynamic array:  $n$  calls to PushBack

# Banker's Method

Dynamic array:  $n$  calls to PushBack

Charge 3 for each insertion: 1 token is the raw cost for insertion.

# Banker's Method

Dynamic array:  $n$  calls to PushBack

Charge 3 for each insertion: 1 token is the raw cost for insertion.

- Resize needed: To pay for moving the elements, use the token that's present on each element that needs to move.

# Banker's Method

Dynamic array:  $n$  calls to PushBack

Charge 3 for each insertion: 1 token is the raw cost for insertion.

- Resize needed: To pay for moving the elements, use the token that's present on each element that needs to move.
- Place one token on the newly-inserted element, and one token  $\frac{\text{capacity}}{2}$  elements prior.

# Dynamic Array Resizing



arr

size: 0   capacity: 0



# Dynamic Array Resizing



arr

size: 0 capacity: 0

PushBack(a)

# Dynamic Array Resizing



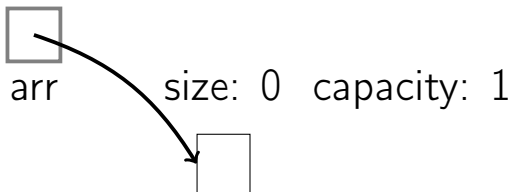
arr

size: 0 capacity: 1



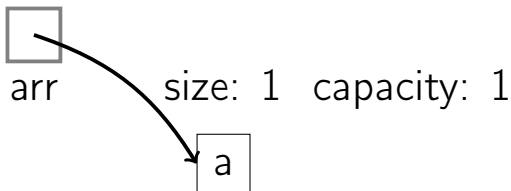
PushBack(a)

# Dynamic Array Resizing



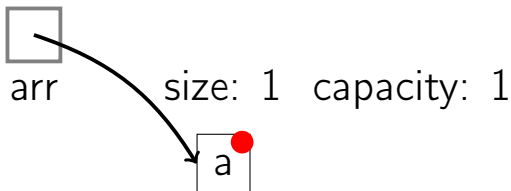
PushBack(a)

# Dynamic Array Resizing



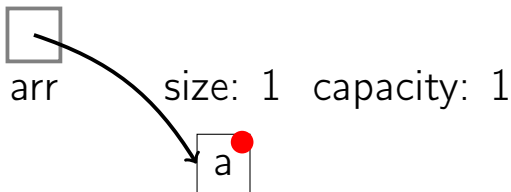
PushBack(a)

# Dynamic Array Resizing

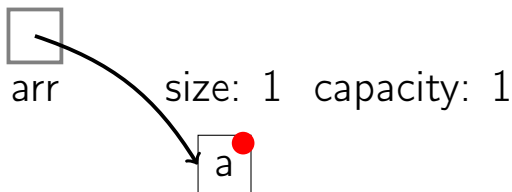


PushBack(a)

# Dynamic Array Resizing

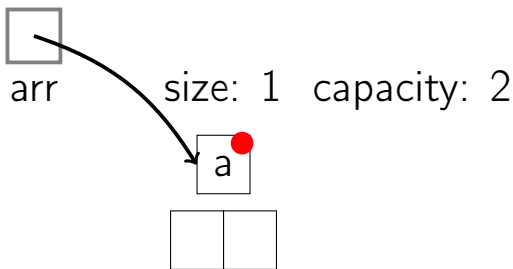


# Dynamic Array Resizing



PushBack(b)

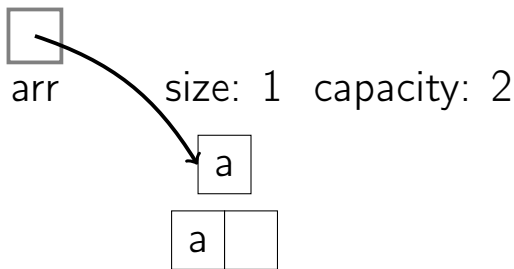
# Dynamic Array Resizing



`PushBack(b)`

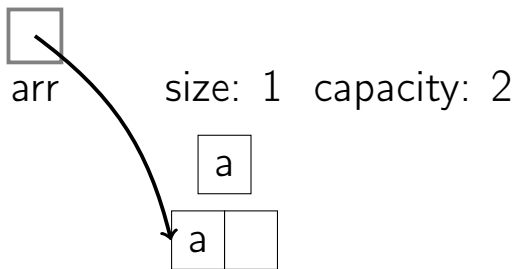


# Dynamic Array Resizing



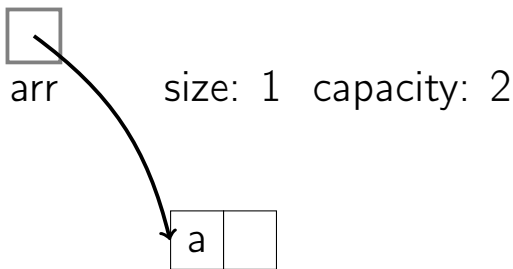
PushBack(b)

# Dynamic Array Resizing



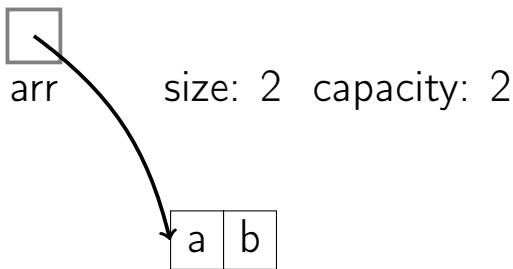
`PushBack(b)`

# Dynamic Array Resizing



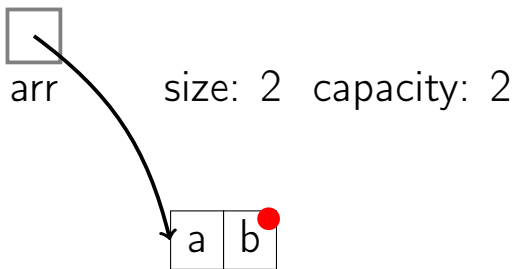
PushBack(b)

# Dynamic Array Resizing



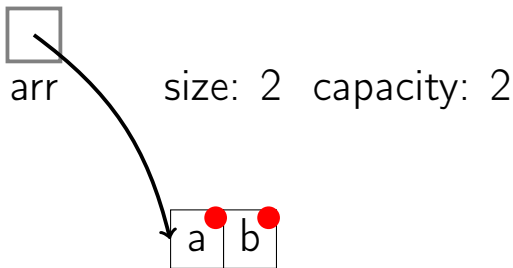
PushBack(b)

# Dynamic Array Resizing



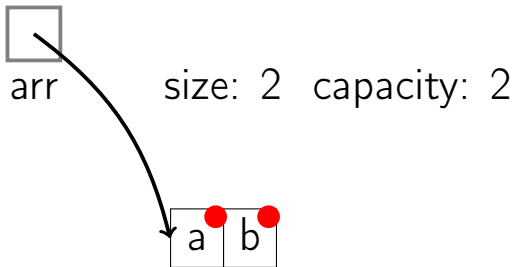
PushBack(b)

# Dynamic Array Resizing



PushBack(b)

# Dynamic Array Resizing

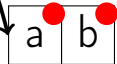


# Dynamic Array Resizing



arr

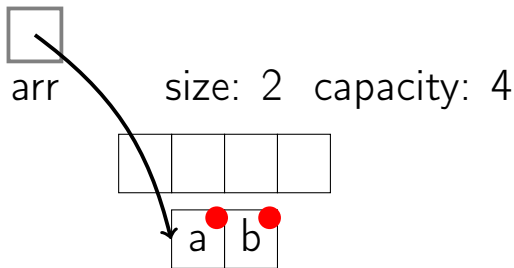
size: 2 capacity: 2



PushBack(c)

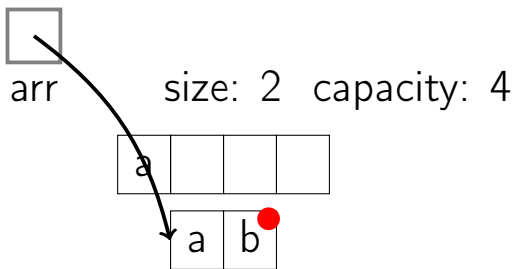


# Dynamic Array Resizing



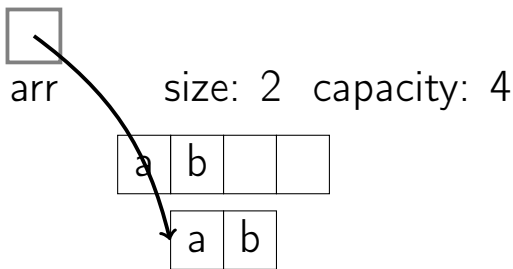
PushBack(c)

# Dynamic Array Resizing



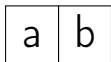
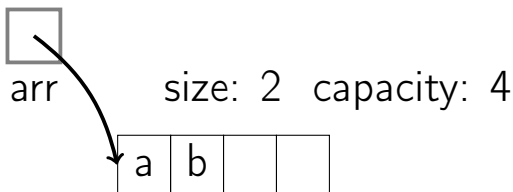
PushBack(c)

# Dynamic Array Resizing



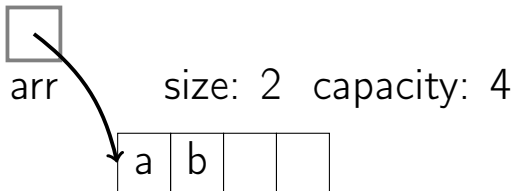
PushBack(c)

# Dynamic Array Resizing



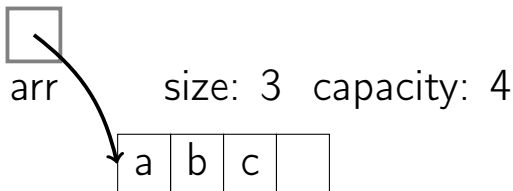
PushBack(c)

# Dynamic Array Resizing



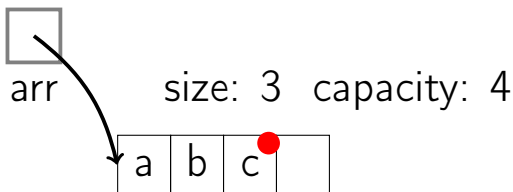
PushBack(c)

# Dynamic Array Resizing



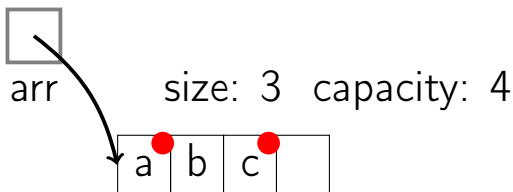
PushBack(c)

# Dynamic Array Resizing



PushBack(c)

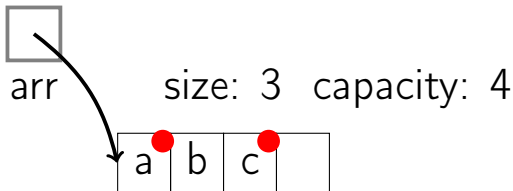
# Dynamic Array Resizing



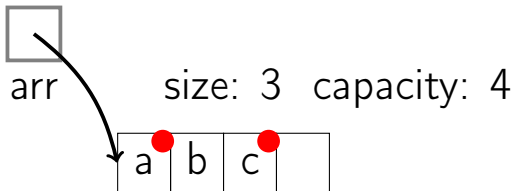
PushBack(c)



# Dynamic Array Resizing

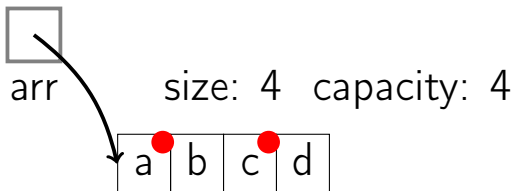


# Dynamic Array Resizing



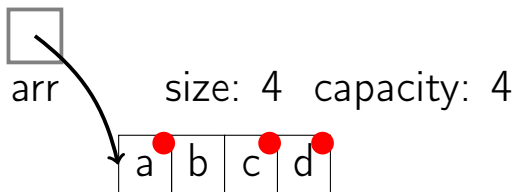
PushBack(d)

# Dynamic Array Resizing



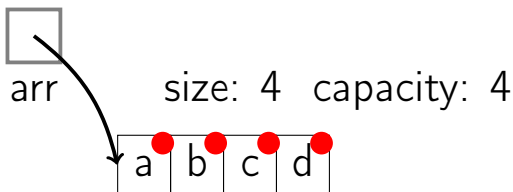
PushBack(d)

# Dynamic Array Resizing



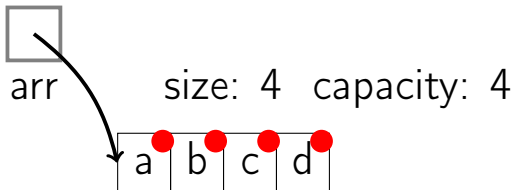
`PushBack(d)`

# Dynamic Array Resizing

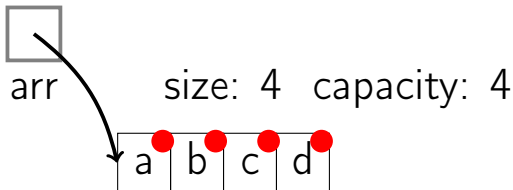


`PushBack(d)`

# Dynamic Array Resizing

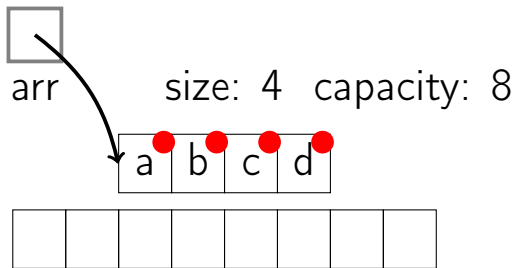


# Dynamic Array Resizing



`PushBack(e)`

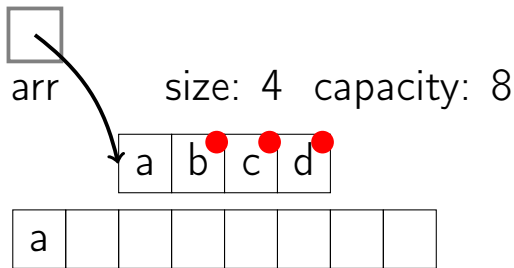
# Dynamic Array Resizing



PushBack(e)

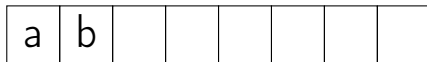
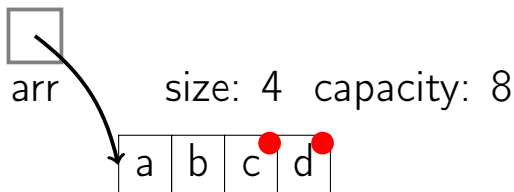


# Dynamic Array Resizing



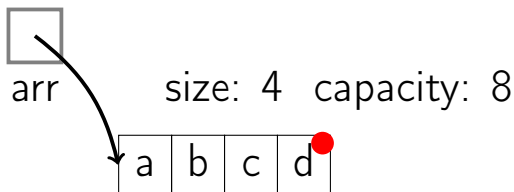
PushBack(e)

# Dynamic Array Resizing



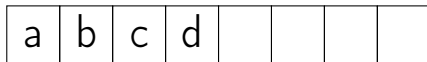
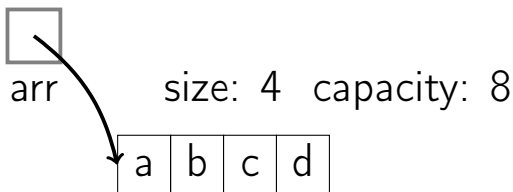
PushBack(e)

# Dynamic Array Resizing



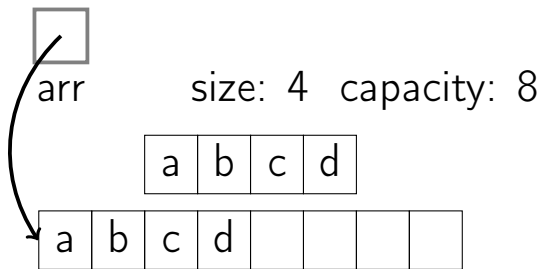
PushBack(e)

# Dynamic Array Resizing



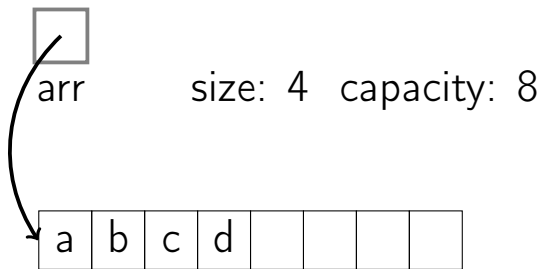
PushBack(e)

# Dynamic Array Resizing



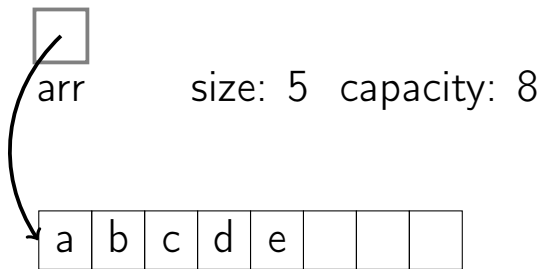
PushBack(e)

# Dynamic Array Resizing



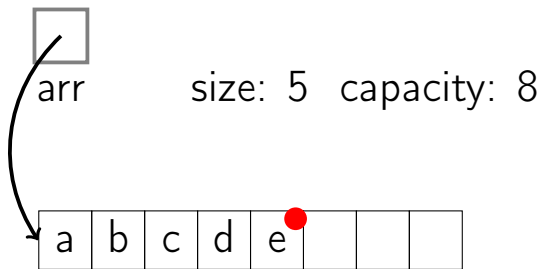
PushBack(e)

# Dynamic Array Resizing



PushBack(e)

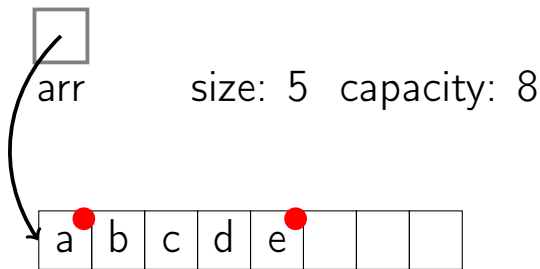
# Dynamic Array Resizing



PushBack(e)

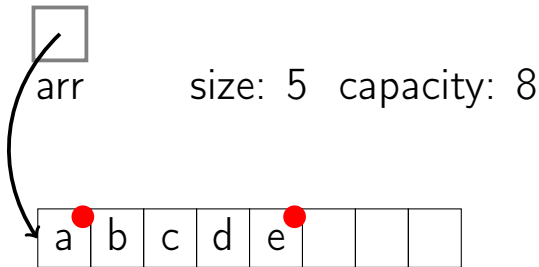


# Dynamic Array Resizing

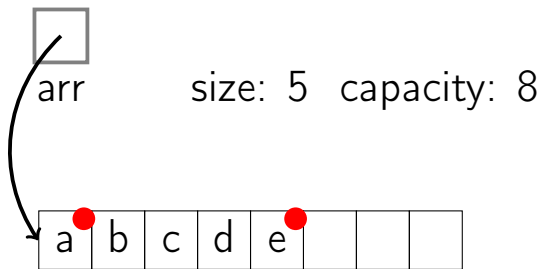


PushBack(e)

# Dynamic Array Resizing



# Dynamic Array Resizing



$O(1)$  amortized cost for each PushBack

# Banker's Method

Dynamic array:  $n$  calls to PushBack

Charge 3 for each insertion. 1 coin is the raw cost for insertion.

- Resize needed: To pay for moving the elements, use the coin that's present on each element that needs to move.
- Place one coin on the newly-inserted element, and one coin  $\frac{\text{capacity}}{2}$  elements prior.

# Outline

- ① Dynamic Arrays
- ② Amortized Analysis—Aggregate Method
- ③ Amortized Analysis—Banker's Method
- ④ Amortized Analysis—Physicist's Method

# Physicist's Method

- Define a *potential function*,  $\Phi$  which maps states of the data structure to integers:

# Physicist's Method

- Define a *potential function*,  $\Phi$  which maps states of the data structure to integers:
  - $\Phi(h_0) = 0$

# Physicist's Method

- Define a *potential function*,  $\Phi$  which maps states of the data structure to integers:
  - $\Phi(h_0) = 0$
  - $\Phi(h_t) \geq 0$



# Physicist's Method

- Define a *potential function*,  $\Phi$  which maps states of the data structure to integers:
  - $\Phi(h_0) = 0$
  - $\Phi(h_t) \geq 0$
- amortized cost for operation  $t$ :  
 $c_t + \Phi(h_t) - \Phi(h_{t-1})$

# Physicist's Method

- Define a *potential function*,  $\Phi$  which maps states of the data structure to integers:
  - $\Phi(h_0) = 0$
  - $\Phi(h_t) \geq 0$
- amortized cost for operation  $t$ :  
$$c_t + \Phi(h_t) - \Phi(h_{t-1})$$

Choose  $\Phi$  so that:

# Physicist's Method

- Define a *potential function*,  $\Phi$  which maps states of the data structure to integers:
  - $\Phi(h_0) = 0$
  - $\Phi(h_t) \geq 0$
- amortized cost for operation  $t$ :  
$$c_t + \Phi(h_t) - \Phi(h_{t-1})$$

Choose  $\Phi$  so that:

- if  $c_t$  is small, the potential increases

# Physicist's Method

- Define a *potential function*,  $\Phi$  which maps states of the data structure to integers:
  - $\Phi(h_0) = 0$
  - $\Phi(h_t) \geq 0$
- amortized cost for operation  $t$ :  
$$c_t + \Phi(h_t) - \Phi(h_{t-1})$$

Choose  $\Phi$  so that:

- if  $c_t$  is small, the potential increases
- if  $c_t$  is large, the potential decreases by the same scale

# Physicist's Method

- The cost of  $n$  operations is:  $\sum_{i=1}^n c_i$

# Physicist's Method

- The cost of  $n$  operations is:  $\sum_{i=1}^n c_i$
- The sum of the amortized costs is:

$$\sum_{i=1}^n (c_i + \Phi(h_i) - \Phi(h_{i-1}))$$

# Physicist's Method

- The cost of  $n$  operations is:  $\sum_{i=1}^n c_i$
- The sum of the amortized costs is:

$$\begin{aligned} & \sum_{i=1}^n (c_i + \Phi(h_i) - \Phi(h_{i-1})) \\ &= c_1 + \Phi(h_1) - \Phi(h_0) + \\ & \quad c_2 + \Phi(h_2) - \Phi(h_1) \cdots + \\ & \quad c_n + \Phi(h_n) - \Phi(h_{n-1}) \end{aligned}$$

# Physicist's Method

- The cost of  $n$  operations is:  $\sum_{i=1}^n c_i$
- The sum of the amortized costs is:

$$\begin{aligned} & \sum_{i=1}^n (c_i + \Phi(h_i) - \Phi(h_{i-1})) \\ &= c_1 + \Phi(h_1) - \Phi(h_0) + \\ & \quad c_2 + \Phi(h_2) - \Phi(h_1) \cdots + \\ & \quad c_n + \Phi(h_n) - \Phi(h_{n-1}) \\ &= \Phi(h_n) - \Phi(h_0) + \sum_{i=1}^n c_i \end{aligned}$$



# Physicist's Method

- The cost of  $n$  operations is:  $\sum_{i=1}^n c_i$
- The sum of the amortized costs is:

$$\begin{aligned} & \sum_{i=1}^n (c_i + \Phi(h_i) - \Phi(h_{i-1})) \\ &= c_1 + \Phi(h_1) - \Phi(h_0) + \\ & \quad c_2 + \Phi(h_2) - \Phi(h_1) \cdots + \\ & \quad c_n + \Phi(h_n) - \Phi(h_{n-1}) \\ &= \Phi(h_n) - \Phi(h_0) + \sum_{i=1}^n c_i \geq \sum_{i=1}^n c_i \end{aligned}$$

# Physicist's Method

Dynamic array:  $n$  calls to PushBack

# Physicist's Method

Dynamic array:  $n$  calls to PushBack

Let  $\Phi(h) = 2 \times \textit{size} - \textit{capacity}$

# Physicist's Method

Dynamic array:  $n$  calls to PushBack

Let  $\Phi(h) = 2 \times \textit{size} - \textit{capacity}$

- $\Phi(h_0) = 2 \times 0 - 0 = 0$

# Physicist's Method

Dynamic array:  $n$  calls to PushBack

Let  $\Phi(h) = 2 \times \textit{size} - \textit{capacity}$

- $\Phi(h_0) = 2 \times 0 - 0 = 0$
- $\Phi(h_i) = 2 \times \textit{size} - \textit{capacity} > 0$   
(since  $\textit{size} > \frac{\textit{capacity}}{2}$ )

# Dynamic Array Resizing

Without resize when adding element  $i$

Amortized cost of adding element  $i$ :

# Dynamic Array Resizing

Without resize when adding element  $i$

Amortized cost of adding element  $i$ :

# Dynamic Array Resizing

Without resize when adding element  $i$

Amortized cost of adding element  $i$ :

$$c_i + \Phi(h_i) - \Phi(h_{i-1})$$



# Dynamic Array Resizing

Without resize when adding element  $i$

Amortized cost of adding element  $i$ :

$$\begin{aligned} & c_i + \Phi(h_i) - \Phi(h_{i-1}) \\ &= 1 + 2 \times \textit{size}_i - \textit{cap}_i - (2 \times \textit{size}_{i-1} - \textit{cap}_{i-1}) \end{aligned}$$

# Dynamic Array Resizing

Without resize when adding element  $i$

Amortized cost of adding element  $i$ :

$$\begin{aligned} & c_i + \Phi(h_i) - \Phi(h_{i-1}) \\ &= 1 + 2 \times \textit{size}_i - \textit{cap}_i - (2 \times \textit{size}_{i-1} - \textit{cap}_{i-1}) \\ &= 1 + 2 \times (\textit{size}_i - \textit{size}_{i-1}) \end{aligned}$$

# Dynamic Array Resizing

Without resize when adding element  $i$

Amortized cost of adding element  $i$ :

$$\begin{aligned} & c_i + \Phi(h_i) - \Phi(h_{i-1}) \\ &= 1 + 2 \times \text{size}_i - \text{cap}_i - (2 \times \text{size}_{i-1} - \text{cap}_{i-1}) \\ &= 1 + 2 \times (\text{size}_i - \text{size}_{i-1}) \\ &= 3 \end{aligned}$$

# Dynamic Array Resizing

With resize when adding element  $i$

# Dynamic Array Resizing

With resize when adding element  $i$

Let  $k = \text{size}_{i-1} = \text{cap}_{i-1}$

# Dynamic Array Resizing

With resize when adding element  $i$

Let  $k = \text{size}_{i-1} = \text{cap}_{i-1}$

Then:

$$\Phi(h_{i-1}) = 2\text{size}_{i-1} - \text{cap}_{i-1} = 2k - k = k$$

# Dynamic Array Resizing

With resize when adding element  $i$

Let  $k = \text{size}_{i-1} = \text{cap}_{i-1}$

Then:

$$\Phi(h_{i-1}) = 2\text{size}_{i-1} - \text{cap}_{i-1} = 2k - k = k$$

$$\Phi(h_i) = 2\text{size}_i - \text{cap}_i = 2(k+1) - 2k = 2$$

# Dynamic Array Resizing

With resize when adding element  $i$

Let  $k = \text{size}_{i-1} = \text{cap}_{i-1}$

Then:

$$\Phi(h_{i-1}) = 2\text{size}_{i-1} - \text{cap}_{i-1} = 2k - k = k$$

$$\Phi(h_i) = 2\text{size}_i - \text{cap}_i = 2(k+1) - 2k = 2$$

Amortized cost of adding element  $i$ :



# Dynamic Array Resizing

With resize when adding element  $i$

Let  $k = \text{size}_{i-1} = \text{cap}_{i-1}$

Then:

$$\Phi(h_{i-1}) = 2\text{size}_{i-1} - \text{cap}_{i-1} = 2k - k = k$$

$$\Phi(h_i) = 2\text{size}_i - \text{cap}_i = 2(k+1) - 2k = 2$$

Amortized cost of adding element  $i$ :

$$c_i + \Phi(h_i) - \Phi(h_{i-1})$$

# Dynamic Array Resizing

With resize when adding element  $i$

Let  $k = \text{size}_{i-1} = \text{cap}_{i-1}$

Then:

$$\Phi(h_{i-1}) = 2\text{size}_{i-1} - \text{cap}_{i-1} = 2k - k = k$$

$$\Phi(h_i) = 2\text{size}_i - \text{cap}_i = 2(k+1) - 2k = 2$$

Amortized cost of adding element  $i$ :

$$\begin{aligned} & c_i + \Phi(h_i) - \Phi(h_{i-1}) \\ &= (\text{size}_i) + 2 - k \end{aligned}$$

# Dynamic Array Resizing

With resize when adding element  $i$

Let  $k = \text{size}_{i-1} = \text{cap}_{i-1}$

Then:

$$\Phi(h_{i-1}) = 2\text{size}_{i-1} - \text{cap}_{i-1} = 2k - k = k$$

$$\Phi(h_i) = 2\text{size}_i - \text{cap}_i = 2(k+1) - 2k = 2$$

Amortized cost of adding element  $i$ :

$$\begin{aligned} & c_i + \Phi(h_i) - \Phi(h_{i-1}) \\ &= (\text{size}_i) + 2 - k \\ &= (k+1) + 2 - k \end{aligned}$$

# Dynamic Array Resizing

With resize when adding element  $i$

Let  $k = \text{size}_{i-1} = \text{cap}_{i-1}$

Then:

$$\Phi(h_{i-1}) = 2\text{size}_{i-1} - \text{cap}_{i-1} = 2k - k = k$$

$$\Phi(h_i) = 2\text{size}_i - \text{cap}_i = 2(k+1) - 2k = 2$$

Amortized cost of adding element  $i$ :

$$\begin{aligned} & c_i + \Phi(h_i) - \Phi(h_{i-1}) \\ &= (\text{size}_i) + 2 - k \\ &= (k+1) + 2 - k \\ &= 3 \end{aligned}$$

# Alternatives to Doubling the Array Size

We could use some different growth factor (1.5, 2.5, etc.).

Could we use a constant amount?

# Cannot Use Constant Amount

If we expand by 10 each time, then:

Let  $c_i$  = cost of  $i$ 'th insertion.

# Cannot Use Constant Amount

If we expand by 10 each time, then:

Let  $c_i$  = cost of  $i$ 'th insertion.

# Cannot Use Constant Amount

If we expand by 10 each time, then:

Let  $c_i$  = cost of  $i$ 'th insertion.

$$c_i = 1 + \left\{ \right.$$



# Cannot Use Constant Amount

If we expand by 10 each time, then:

Let  $c_i$  = cost of  $i$ 'th insertion.

$$c_i = 1 + \begin{cases} i - 1 & \text{if } i - 1 \text{ is a multiple of } 10 \end{cases}$$

# Cannot Use Constant Amount

If we expand by 10 each time, then:

Let  $c_i$  = cost of  $i$ 'th insertion.

$$c_i = 1 + \begin{cases} i - 1 & \text{if } i - 1 \text{ is a multiple of } 10 \\ 0 & \text{otherwise} \end{cases}$$

# Cannot Use Constant Amount

If we expand by 10 each time, then:

Let  $c_i$  = cost of  $i$ 'th insertion.

$$c_i = 1 + \begin{cases} i - 1 & \text{if } i - 1 \text{ is a multiple of 10} \\ 0 & \text{otherwise} \end{cases}$$

$$\frac{\sum_{i=1}^n c_i}{n}$$

# Cannot Use Constant Amount

If we expand by 10 each time, then:

Let  $c_i$  = cost of  $i$ 'th insertion.

$$c_i = 1 + \begin{cases} i - 1 & \text{if } i - 1 \text{ is a multiple of } 10 \\ 0 & \text{otherwise} \end{cases}$$

$$\frac{\sum_{i=1}^n c_i}{n} = \frac{n + \sum_{j=1}^{(n-1)/10} 10j}{n}$$

# Cannot Use Constant Amount

If we expand by 10 each time, then:

Let  $c_i$  = cost of  $i$ 'th insertion.

$$c_i = 1 + \begin{cases} i - 1 & \text{if } i - 1 \text{ is a multiple of } 10 \\ 0 & \text{otherwise} \end{cases}$$

$$\frac{\sum_{i=1}^n c_i}{n} = \frac{n + \sum_{j=1}^{(n-1)/10} 10j}{n} = \frac{n + 10 \sum_{j=1}^{(n-1)/10} j}{n}$$

# Cannot Use Constant Amount

If we expand by 10 each time, then:

Let  $c_i$  = cost of  $i$ 'th insertion.

$$c_i = 1 + \begin{cases} i - 1 & \text{if } i - 1 \text{ is a multiple of } 10 \\ 0 & \text{otherwise} \end{cases}$$

$$\begin{aligned} \frac{\sum_{i=1}^n c_i}{n} &= \frac{n + \sum_{j=1}^{(n-1)/10} 10j}{n} = \frac{n + 10 \sum_{j=1}^{(n-1)/10} j}{n} \\ &= \frac{n + 10O(n^2)}{n} \end{aligned}$$

# Cannot Use Constant Amount

If we expand by 10 each time, then:

Let  $c_i$  = cost of  $i$ 'th insertion.

$$c_i = 1 + \begin{cases} i - 1 & \text{if } i - 1 \text{ is a multiple of } 10 \\ 0 & \text{otherwise} \end{cases}$$

$$\begin{aligned} \frac{\sum_{i=1}^n c_i}{n} &= \frac{n + \sum_{j=1}^{(n-1)/10} 10j}{n} = \frac{n + 10 \sum_{j=1}^{(n-1)/10} j}{n} \\ &= \frac{n + 10O(n^2)}{n} = \frac{O(n^2)}{n} \end{aligned}$$

# Cannot Use Constant Amount

If we expand by 10 each time, then:

Let  $c_i$  = cost of  $i$ 'th insertion.

$$c_i = 1 + \begin{cases} i - 1 & \text{if } i - 1 \text{ is a multiple of } 10 \\ 0 & \text{otherwise} \end{cases}$$

$$\begin{aligned} \frac{\sum_{i=1}^n c_i}{n} &= \frac{n + \sum_{j=1}^{(n-1)/10} 10j}{n} = \frac{n + 10 \sum_{j=1}^{(n-1)/10} j}{n} \\ &= \frac{n + 10O(n^2)}{n} = \frac{O(n^2)}{n} = O(n) \end{aligned}$$



# Summary

- Calculate amortized cost of an operation in the context of a sequence of operations.

# Summary

- Calculate amortized cost of an operation in the context of a sequence of operations.
- Three ways to do analysis:

# Summary

- Calculate amortized cost of an operation in the context of a sequence of operations.
- Three ways to do analysis:
  - Aggregate method (brute-force sum)

# Summary

- Calculate amortized cost of an operation in the context of a sequence of operations.
- Three ways to do analysis:
  - Aggregate method (brute-force sum)
  - Banker's method (tokens)

# Summary

- Calculate amortized cost of an operation in the context of a sequence of operations.
- Three ways to do analysis:
  - Aggregate method (brute-force sum)
  - Banker's method (tokens)
  - Physicist's method (potential function,  $\Phi$ )

# Summary

- Calculate amortized cost of an operation in the context of a sequence of operations.
- Three ways to do analysis:
  - Aggregate method (brute-force sum)
  - Banker's method (tokens)
  - Physicist's method (potential function,  $\Phi$ )
- Nothing changes in the code: runtime analysis only.