CS261 Data Structures

Dynamic Arrays

Arrays: Pros and Cons

- Pro: only core data structure designed to hold a collection of elements
- Pro: random access: can quickly get to any element → O(1)
- Con: fixed size:
  - Maximum number of elements must be specified when created

Dynamic Array (Vector or ArrayList)

- The dynamic array (called Vector or ArrayList in Java, same thing, different API) gets around this by ***encapsulating*** a ***partially filled array that can grow when filled***

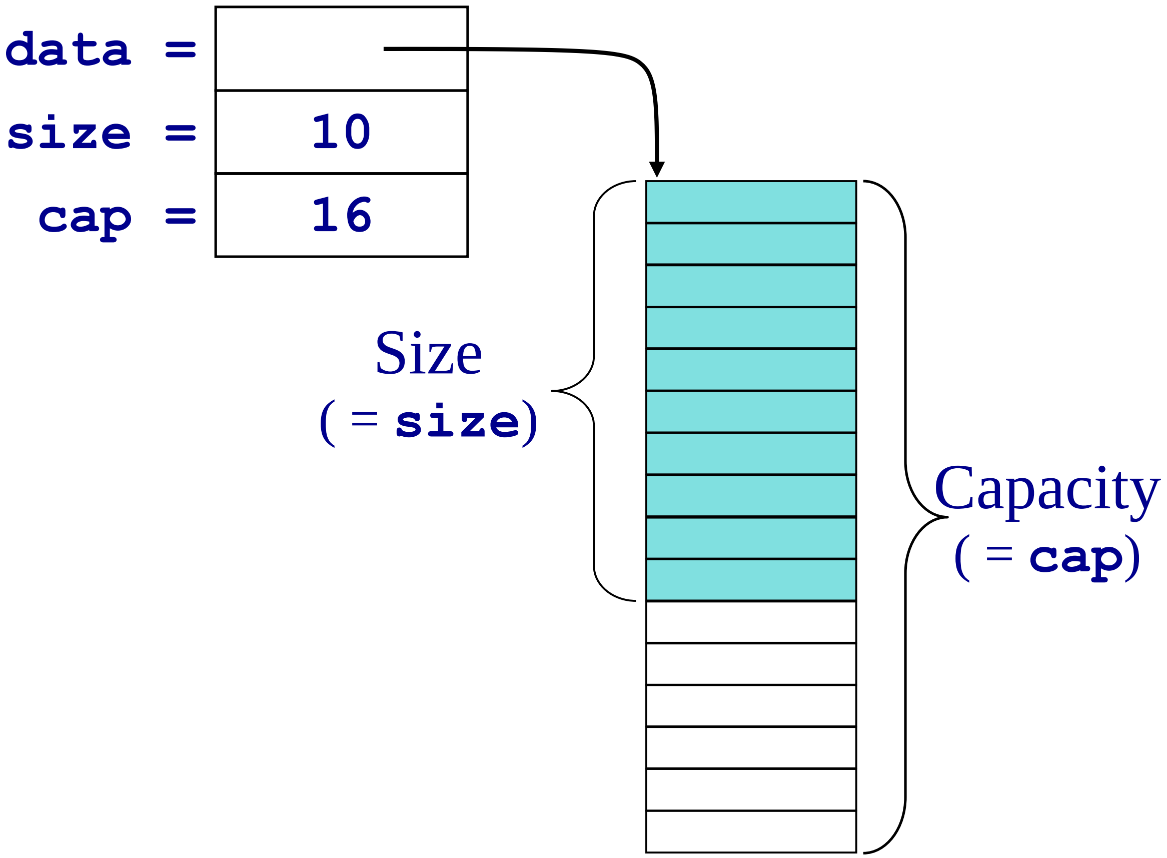
- Hide memory management details behind a simple API

- Is still randomly accessible, but now it grows as necessary

Size and Capacity

- Unlike arrays, a dynamic array can change its capacity
- *Size* is logical collection size:
  - Current number of elements in the dynamic array
  - What the programmer thinks of as the size of the collection
  - Managed by an internal data value
- *Capacity* is physical array size: # of elements it can hold before it must resize

Partially Filled Dynamic Array



Adding an element

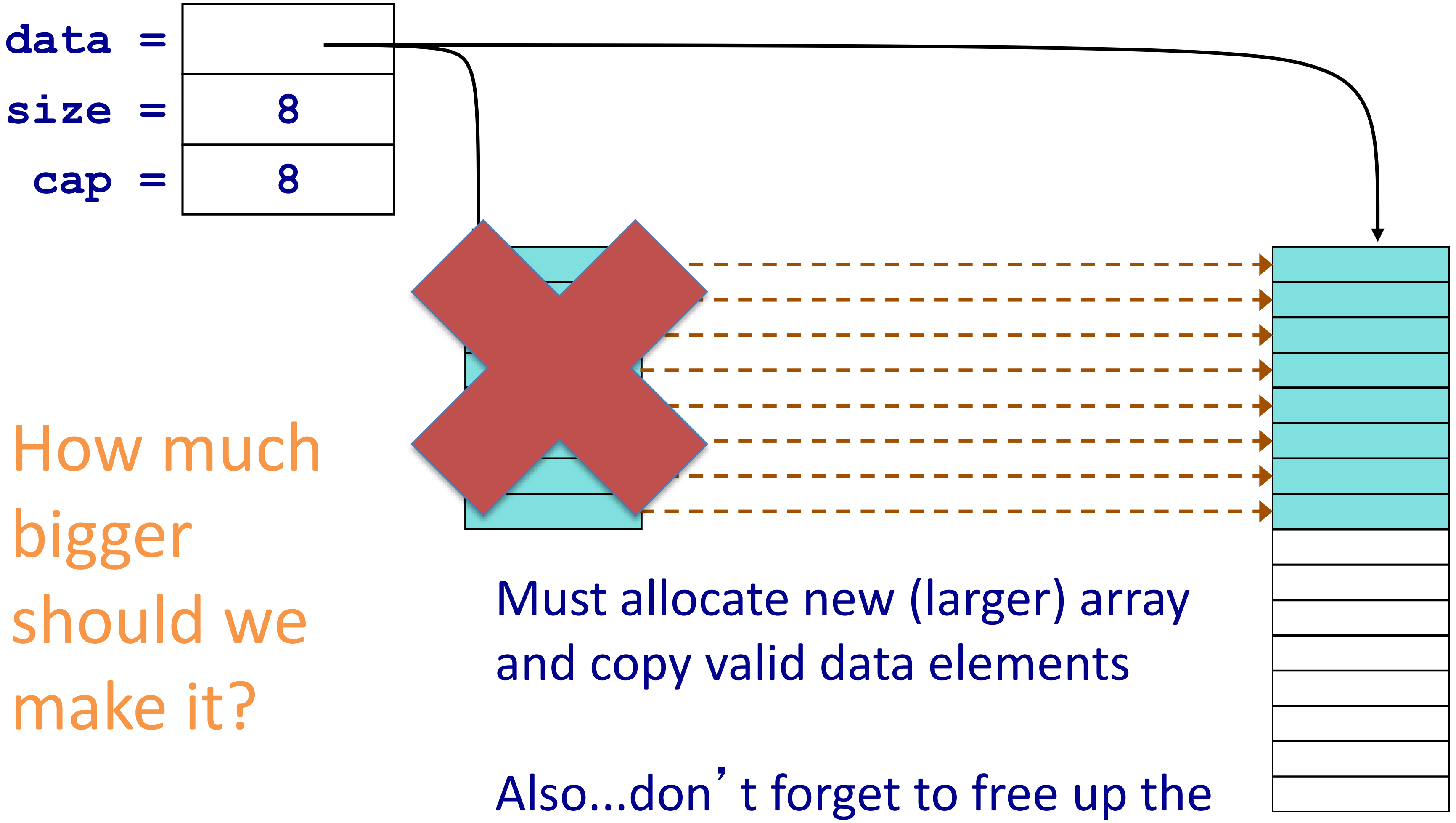
- Adding an element to end is usually easy — just put new value at end and increment the (logical) size

- What happens when size reaches capacity?

Set Capacity: Reallocate and Copy (animation)

Before reallocation:

After reallocation:



How much bigger should we make it?

Must allocate new (larger) array and copy valid data elements

Also...don’t forget to free up the old array

Adding to Middle

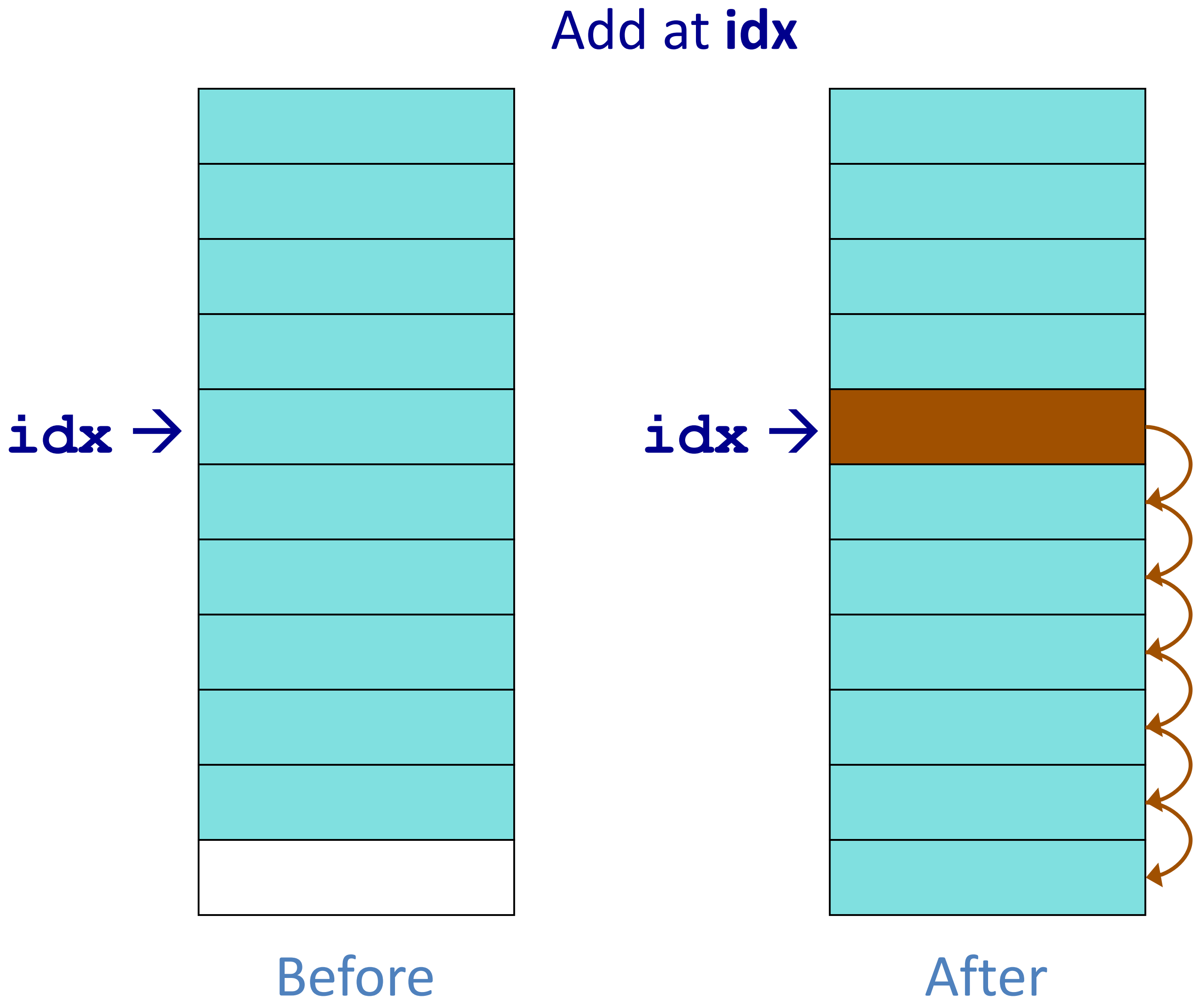
- Adding an element to middle can also force reallocation (if the current size is equal to capacity)
- But will ALWAYS require that elements be moved to make space
  - Our partially filled array should not have gaps so that we always know where the next element should go
- Adding to anywhere other than end is therefore O(*n*) worst case

Adding to Middle (cont.)

Must make space for new value

Be Careful!

Loop from bottom up while copying data



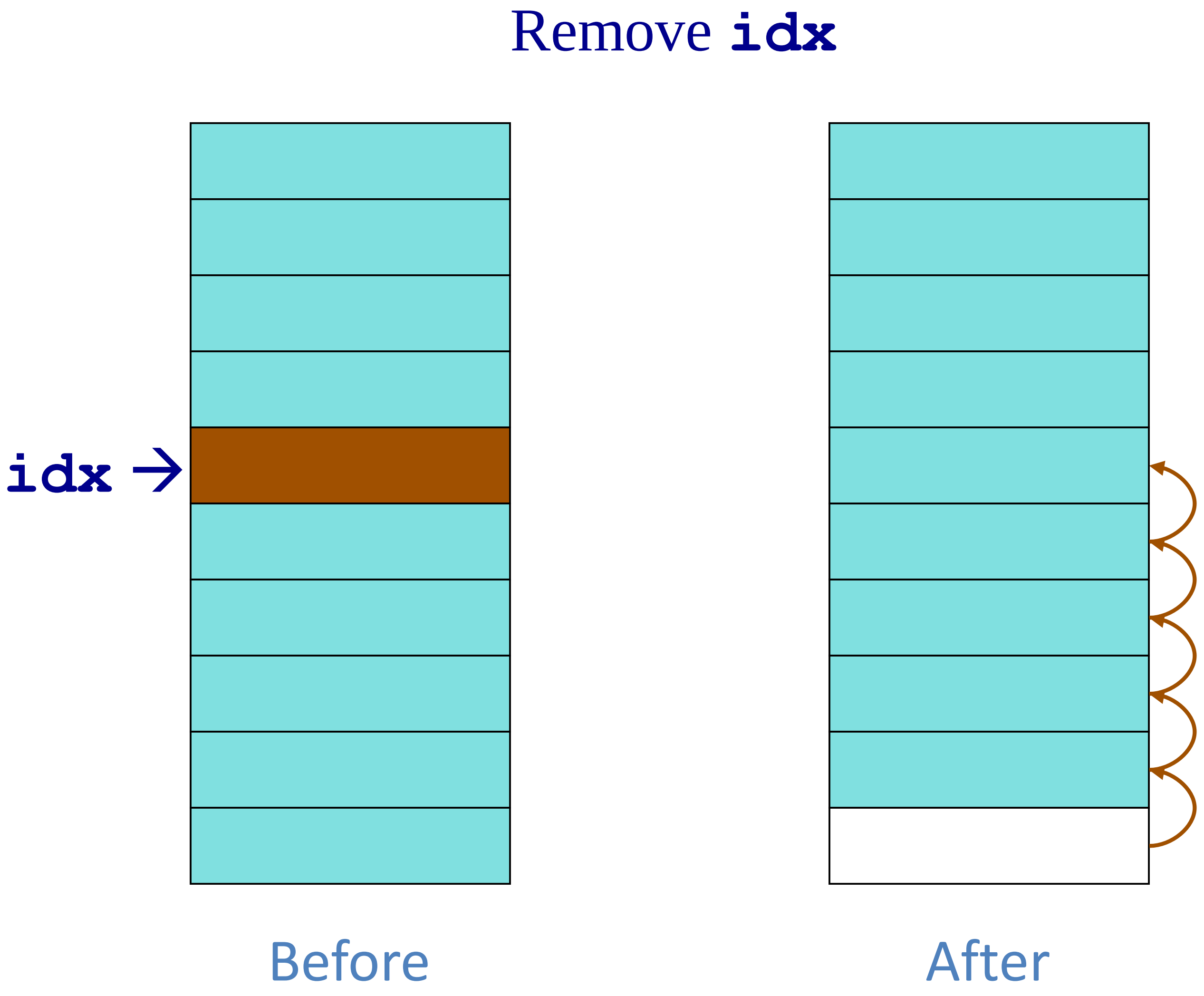
Removing an Element

- Removing an element will also require “sliding over” to delete the value
  - We want to maintain a contiguous chunk of data so we always know where the next element goes and can put it there quickly!
- Therefore is O(*n*) worst case

Remove Element

Remove also requires a loop

This time, should it be from top (e.g. at idx) or bottom?



Side Note

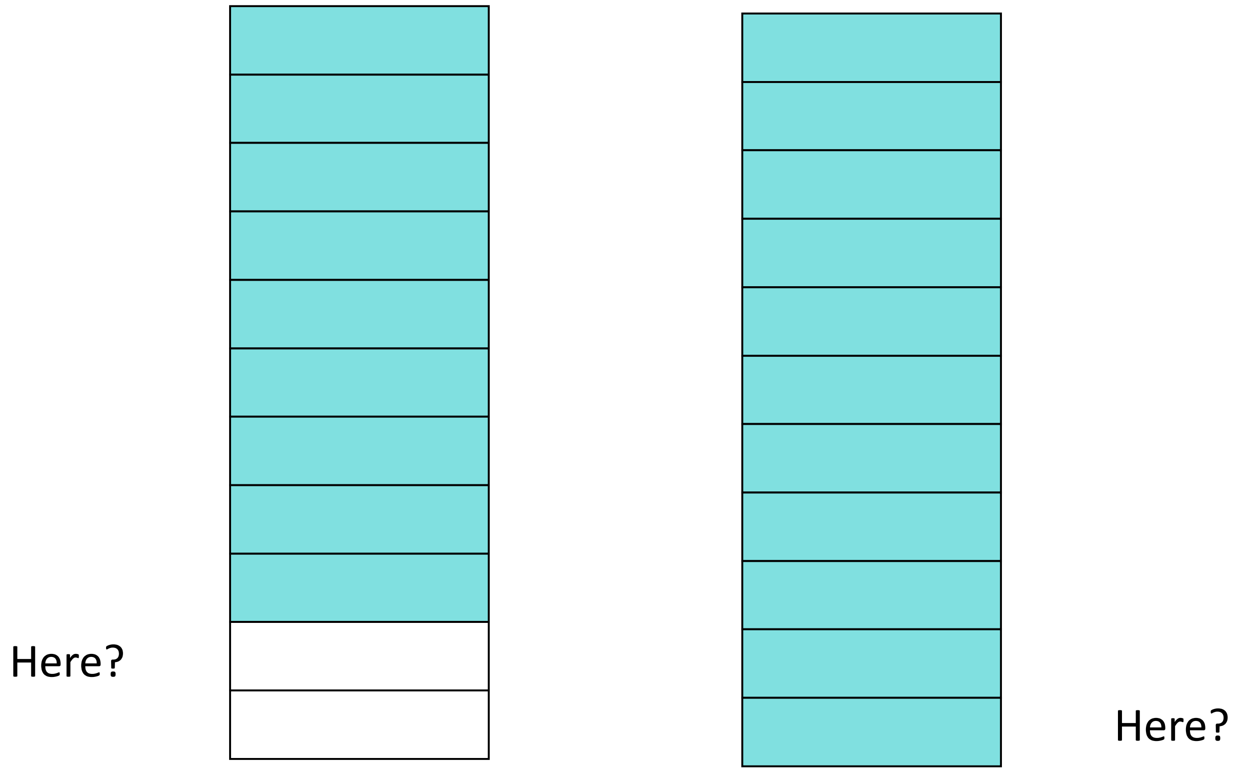
- realloc() can be used in place of malloc() to do resizing and *may* avoid ‘copying’ elements if possible
  - It’s still O(n) when it fails to enlarge the current array!
- For this class, use malloc only (so you’ll have to copy elements on a resize)

Something to think about…

- In the long term, are there any potential problems with the dynamic array?
  - hint: imagine adding MANY elements in the long term and potentially removing many of them.

Amortized Analysis

- What's the cost of adding an element to the end of the array?



Amortized Analysis

- To analyze an algorithm in which the worst case only occurs seldomly, we must perform an amortized analysis to get the average performance
- We’ll use the Accounting or ***Banker’s Method***

Banker's Method

- Assign a cost c'_i to each operation]
- When you perform the operation, if the actual cost c_i , is less, then we save the credit $c'_i - c_i$ to hand out to future operations
- Otherwise, if the actual cost is more than the assigned cost, we borrow from the saved balance
- For n operations, the sum of the total assigned costs must be >= sum of actual costs

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$$

Example – Adding to Dynamic Array

Add Element	Old Capacity	New Capacity	Copy Count	c'_i	c_i	b_i
1	1	1	0	3	1	(3-1) = 2
2	1	2	1	3	(1+1) = 2	(5-2) = 3
3	2	4	2	3	(2+1) = 3	(6-3) = 3
4	4	4			1	(6-1) = 5
5	4	8			(	
6	8	8				
7	8	8				
8	8	8	0	3		
9	8	16	8	3		
10	16	16	0	3		

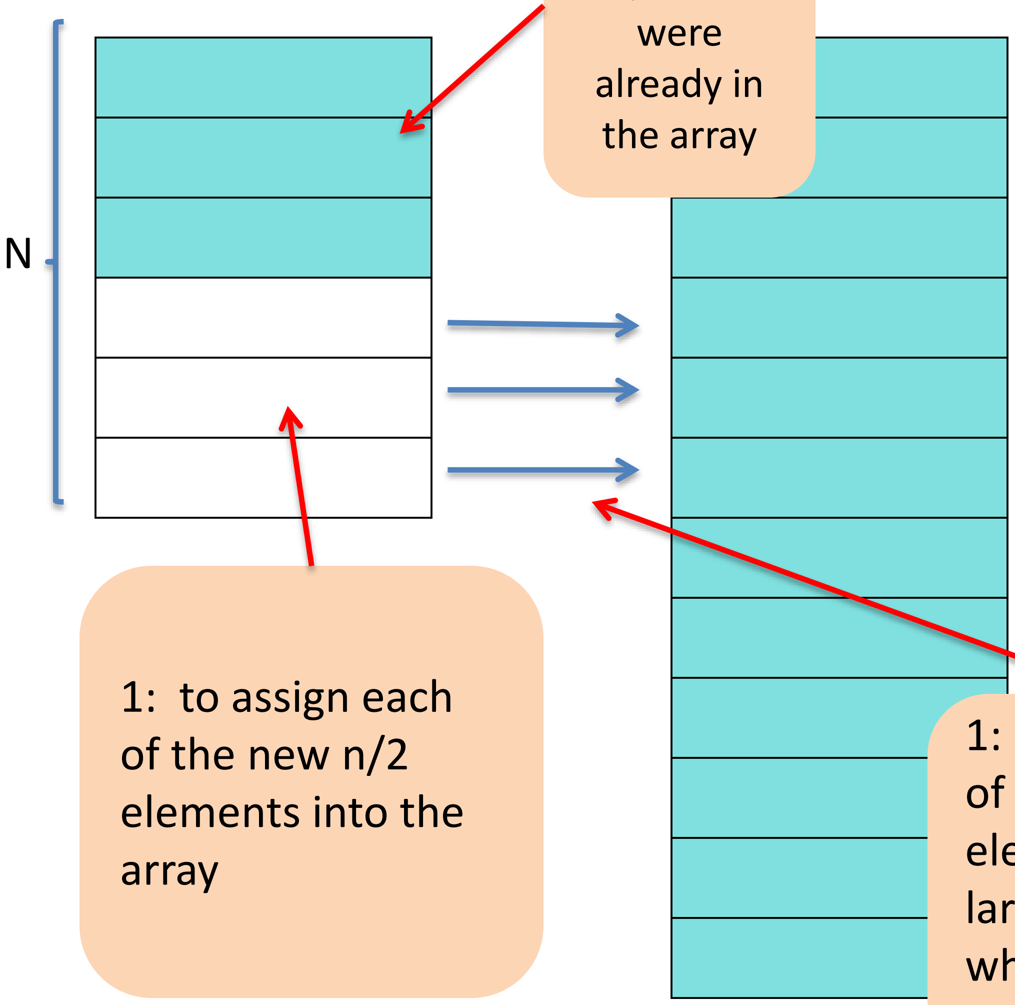
c_i = actual cost = insert (1) + copy cost (1)

b_i = bank account $_i$
=bankaccount$_{i-1}$
+ current deposit
- actual cost
= $(b_{i-1} + c'_i) - c_i$

We say the add() operation is therefore O(1$^+$) – amortized constant cost!

Why do we bank a cost of 3 ?

Imagine you're starting with a partially filled array of size n. It already has n/2 elements in it. For each element you add we'll put a cost of 3 in the bank



1: to copy the other n/2 that were already in the array

1: to assign each of the new n/2 elements into the array

1: to copy each of the new n/2 elements into a larger array when it's time to resize