COMP2402

Abstract Data Types and Algorithms

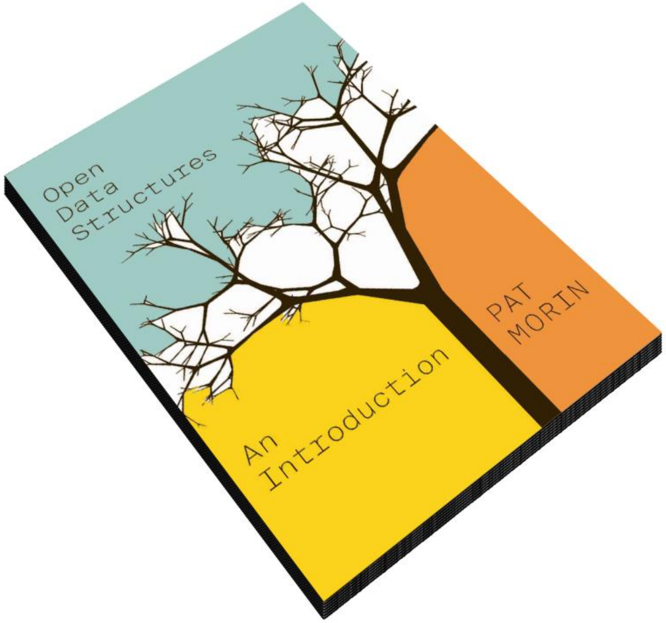
Introduction to Amortized Analysis

Reading Assignment

Open Data Structures in Java

by Pat Morin

Chapter 2.1



Time Complexity Analysis

What were the Worst-Case Time Complexities of:

`add(i, o)`? `push(o)`?
`remove(i)`? `pop()`?

What were the Worst-Case Scenarios?

When the Capacity of the Backing Array was Too Small to Support the Insertion of Another Element or Too Large (i.e., Inefficient w.r.t the Amount of Data), the `resize()` method was Called

What Time Complexity is Associated with Resizing?

Time Complexity for Resizing

```
protected void resize() {
  T[] resized = factory.newArray(Math.max(size*2,1));
  for (int i = 0; i < size; i++) {
    resized[i] = data[i];
  }
  data = resized;
}
```

Every Call to the `resize()` Method for an Instance Containing `n` Elements of Data:

1. Allocates a New Array of a Size that is Twice `n`
2. Copies `n` Elements from Old Array to New Array

this entails Linear Time Complexity – O(`n`)

Time Complexity Analysis

in this case, **Worst-Case Time Complexity** is a **Poor Guide**

Why?

Time Complexity Analysis

in this case, Worst-Case Time Complexity is a Poor Guide
Why?

Too ~~Pessimistic~~ Unrealistic!

the Worst-Case Scenario Does Not Occur Often
(two consecutive calls to add cannot* both require a call to resize)
*except if the backing array is very small
(furthermore, the actual number of calls to resize is typically much less)

How Often is the Resize Method Called?

Lemma

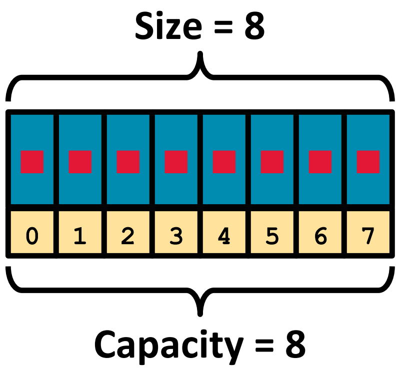
(a Minor Theorem used in a Larger Proof)

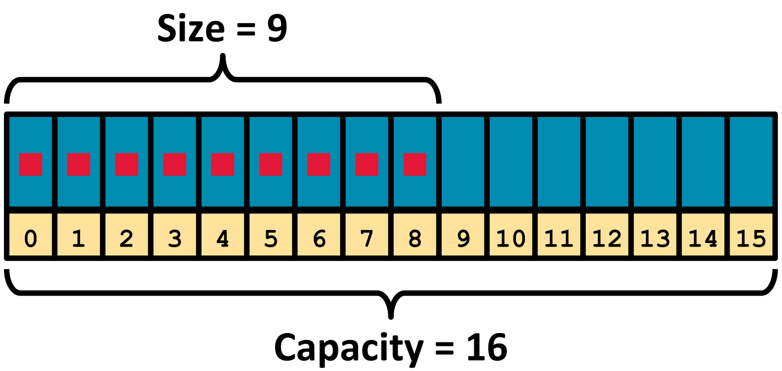
If an Empty List (w/ a Backing Array) is Created, After Any Non-Empty Sequence of m Calls to `add` or `remove` is performed, the Total Time Spent Resizing is O(m)

Amortized Analysis

Suppose you just Finished Resizing the Backing Array

e.g.,

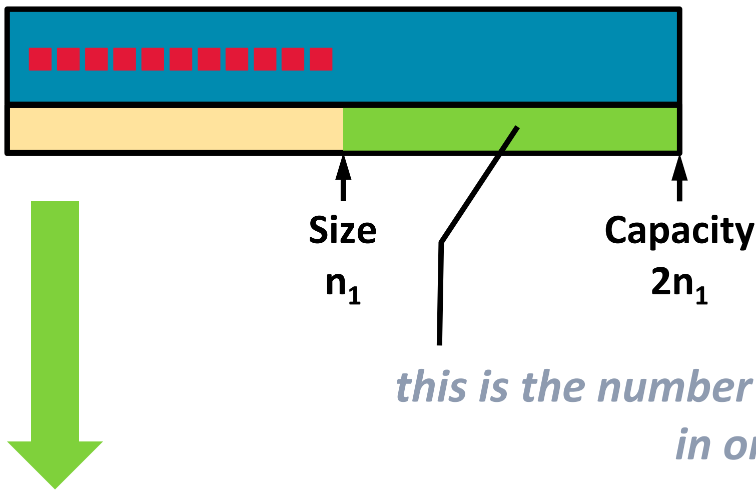





```
size > data.length?
      8 > 8?
       false
         ↓
      insert()
         ↓
size > data.length?
      9 > 8?
       true
         ↓
      resize()
```

How Many Operations Before you Resize Again?

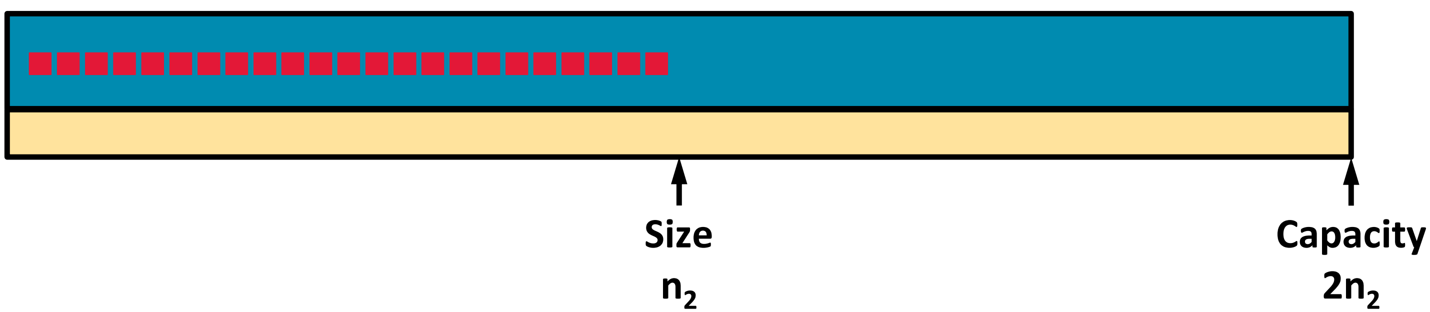
"Grow"ing a Backing Array



How Many Adds Before an Array must "Grow"?

n_1

this is the number of elements that need to be "add"ed in order to cause a "resize"

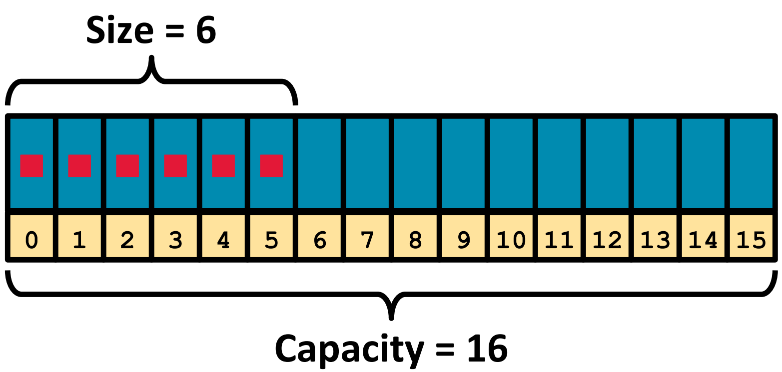


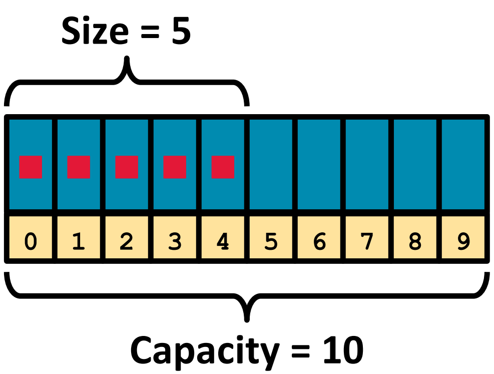
if n_2 is the number of elements in the array after resizing, then $n_2/2$ elements would need to have been added to cause the array to "grow" to the new capacity

Amortized Analysis, Revisited

Suppose you just Finished Resizing the Backing Array

e.g.,

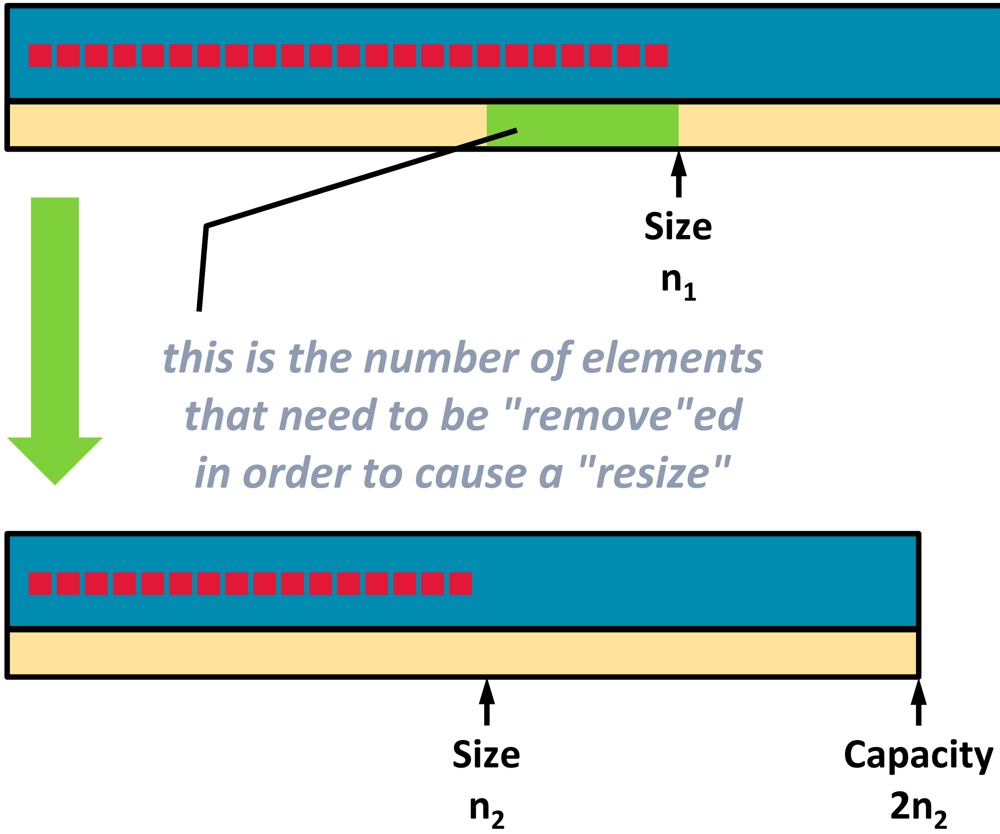





```
data.length >= 3*size?
16 >= 3*6?
false
↓
remove()
↓
data.length >= 3*size?
16 >= 3*5?
true
↓
resize()
```

How Many Operations Before you Resize Again?

"Shrink"ing a Backing Array



How Many Removes Before an Array must "Shrink"?

$n_1/3$

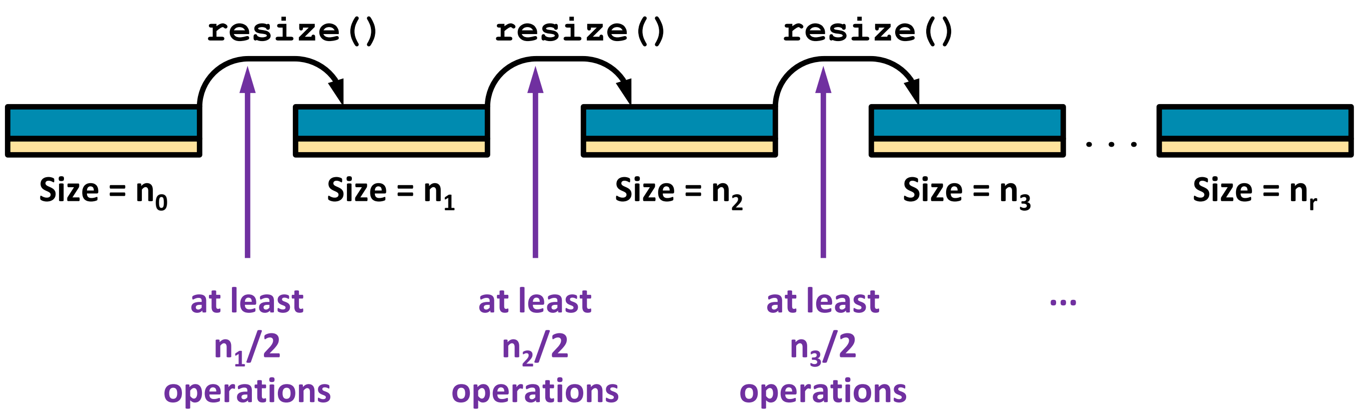
if n_2 is the number of elements after resizing, and n_2 is $2n_1/3$, then $n_1/3 = n_2/2$ elements would need to have been removed to cause the array to "shrink"

Amortized Analysis

Whether the Backing Array is Growing or Shrinking (or a Combination of Both) the Number of Adds/Removes as Array is Resized to Accomodate Size n_i is At Least* $n_i/2$

* technically this should be $n_i/2 - 1$ to account for the special case where the backing array is very small, but this won't change the final result

If m is the Total Number of Add/Remove Calls… and If there are r Calls to Resize…



Amortized Analysis

$$\frac{n_1}{2} + \frac{n_2}{2} + \frac{n_3}{2} + \cdots \frac{n_r}{2} \leq m$$

$$\sum_{i=1}^{r} \frac{n_i}{2} \leq m$$

Interested in the Total Time Spent During All Resize Calls

(i.e., what is the time complexity of that?)

What is the Time Complexity of One Resize (to size n)?

Amortized Analysis

$$\frac{n_1}{2} + \frac{n_2}{2} + \frac{n_3}{2} + \cdots \frac{n_r}{2} \leq m$$

$$\sum_{i=1}^{r} \frac{n_i}{2} \leq m$$

Interested in the Total Time Spent During All Resize Calls

(i.e., what is the time complexity of that?)

What is the Time Complexity of One Resize (to size n)?

O(n)

Sum Over All Calls to Resize….

Amortized Analysis

Find:

$$\sum_{i=1}^{r} O(n_i) \stackrel{?}{=}$$

Since:

$$\sum_{i=1}^{r} \frac{n_i}{2} \leq m$$

Amortized Analysis

Find:

$$\sum_{i=1}^{r} O(n_i) \stackrel{?}{=}$$

Since:

$$\sum_{i=1}^{r} \frac{n_i}{2} \leq m$$

$$\sum_{i=1}^{r} n_i \leq 2m$$

$$\sum_{i=1}^{r} O(n_i) \leq O(2m) = O(m)$$

Amortized Analysis

Lemma

If an Empty List (w/ a Backing Array) is Created, After Any Non-Empty Sequence of m Calls to `add` or `remove` is Performed, the Total Time Spent Resizing is O(m)

Theorem

Array-Backed Implementation of List Interface Supports:

Get and Set in Constant Time*

Add and Remove in Linear Time*

*disregarding the cost associated with the resize operation

and Any Sequence of m Add and Remove Operations Incurs Resizing Costs on the Order O(m)