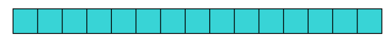
ArrayLists

David Kauchak
cs201
Spring 2014

Extendable array



Arrays store data in sequential locations in memory

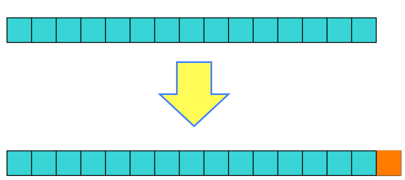
Elements are accessed via their index
- Access of particular indices is O(1)

Say we want to implement an array that supports *add* (i.e. *addToBack*)
- ArrayList or Vector in Java
- lists in Python, perl, Ruby, …

How can we do it?

Extensible array

Idea 1: Each time we call *add*, create a new array one element large, copy the data over and add the element

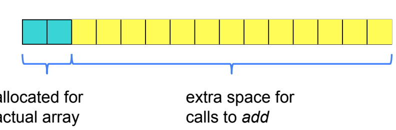


Running time: O(n)

Extensible array

Idea 2: Allocate extra, unused memory and save room to add elements

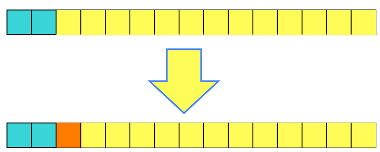
For example: new ArrayList(2)



Extensible array

Idea 2: Allocate extra, unused memory and save room to add elements

Adding an item:

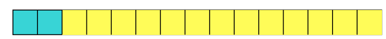


Running time: O(1)   Problems?

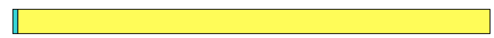
Extensible array

Idea 2: Allocate extra, unused memory and save room to add elements

How much extra space do we allocate?



Too little, and we might run out (e.g. add 15 items)

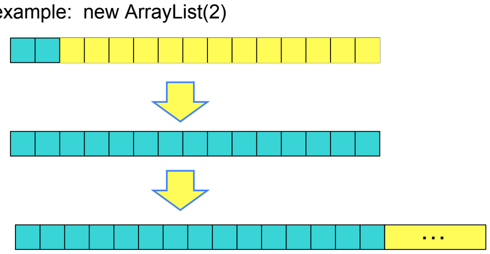


Too much, and we waste lots of memory   Ideas?

Extensible array

Idea 3: Allocate some extra memory and when it fills up, allocate some more and copy

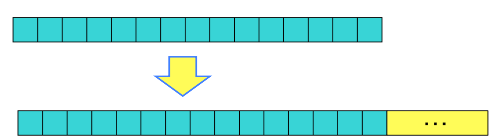
For example: new ArrayList(2)



Extensible array

Idea 3: Allocate some extra memory and when it fills up, allocate some more and copy

For example: new ArrayList(2)

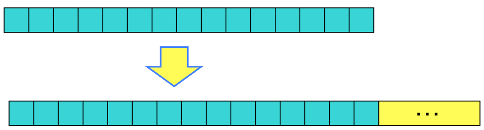


Running time: O(n)

Extensible array

Idea 3: Allocate some extra memory and when it fills up, allocate some more and copy

For example: new ArrayList(2)



How much extra memory should we allocate?

Extensible array

Idea 3: Allocate some extra memory and when it fills up, allocate some more and copy

For example: new ArrayList(2)

What is the best case running time of add? O(1)

What is the worst case running time of add? O(n)

Can we bound this tighter?

Extensible array

Challenge: most of the calls to *add* will be O(1)

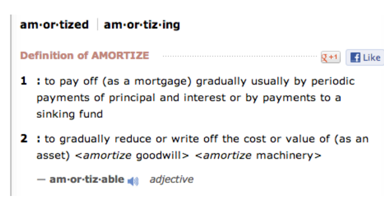
How else might we talk about runtime?

What is the ***average*** running time of *add* in the ***worst case***?

Note this is different than the *average-case* running time

Amortized analysis

What does "amortize" mean?



am·or·tized | **am·or·tiz·ing**

Definition of AMORTIZE

1 **:** to pay off (as a mortgage) gradually usually by periodic payments of principal and interest or by payments to a sinking fund

2 **:** to gradually reduce or write off the cost or value of (as an asset) <*amortize* goodwill> <*amortize* machinery>

— **am·or·tiz·able** *adjective*

Amortized analysis

There are many situations where the worst case running time is bad

However, if we average the operations over *n* operations, the average time is more reasonable

This is called *amortized* analysis

- This is different than average-case running time, which requires reasoning about the input/situations that the method will be called
- The worse case running time doesn't change

Amortized analysis

Many approaches for calculating the amortized analysis

Aggregate method

- figure out the big-O runtime for a sequence of *n* calls
- divide by *n* to get the average run-time per call

Amortized analysis

Assume we start with an empty array with 1 location. What is the cost to insert n items?

$$\text{total\_cost(n)} = \text{basic\_cost(n)} + \text{double\_cost(n)}$$

CHALKBOARD ☺

Amortized analysis

Assume we start with an empty array with 1 location. What is the cost to insert n items?

$$\text{total\_cost(n)} = \text{basic\_cost(n)} + \text{double\_cost(n)}$$

$$\text{basic\_cost(n)} = O(n) \qquad \text{double\_cost(n)} \leq 1+2+4+8+16+\ldots+n = 2n$$

$$\text{total\_cost(n)} = O(n)$$

amortized *O(1)*

Amortized analysis vs. worse case

What is the worst case for *add*?

- Still O(n)
- If you have an application that needs it to be O(1), this implementation **will not work!**

amortized analysis give you the cost of *n* operations (i.e. average cost) **not** the cost of any individual operation

Extensible arrays

What if instead of doubling the array, we increase the array by a fixed amount (call it *k*) each time

Is the amortized run-time still O(1)?

- No!
- Why?

Amortized analysis

Consider the cost of *n* insertions for some constant *k*

$$\text{total\_cost(n)} = \text{basic\_cost(n)} + \text{double\_cost(n)}$$

$$\text{basic\_cost(n)} = O(n)$$

$$\begin{aligned}\text{double\_cost(n)} &= \text{k+2k+3k+4k+5k+...+n}\\ &= \sum_{i=1}^{n/k} ki\\ &= \text{k}\sum_{i=1}^{n/k} i\\ &= \text{k}\frac{\frac{n}{k}\left(\frac{n}{k}+1\right)}{2} = O(n^2)\end{aligned}$$

Amortized analysis

Consider the cost of *n* insertions for some constant *k*

$$\begin{aligned}\text{total\_cost(n)} &= O(n) + O(n^2)\\ &= O(n^2)\end{aligned}$$

amortized *O(n)!*