Dynamic arrays and amortized analysis

Lecture 02.02 by Marina Barsky

It is not possible to always know in advance how big an array should be

If we add an element past the capacity of the array:

Bad things happen:

- Java: Array index out of bound
- Python: List index out of range
- C: No warnings, total corruption of program memory

Dynamic allocation of space

- We keep track of the number of elements in the array
- If we need more space - we allocate new space and transfer data from an old array
- This requires O(n) operations to copy the data

```
int myArray[100];
int[] newArray = new int[200];
System.arraycopy(myArray, newArray, 0,100);
```

```
int *my_array = malloc (100*size_of(int));
my_array = realloc(200*size_of(int))
```

A new data structure

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:

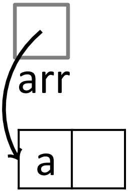
Data structure that supports the same operations as a static array, but does not limit the number of elements that it can hold.

Dynamic array

Contains 3 variables:

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

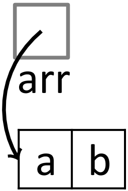
Dynamic Array: Resizing



size: 1
capacity: 2

`add(a)`

Dynamic Array: Resizing

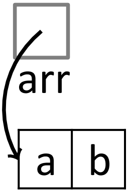


size: 2

capacity: 2

add(b)

Dynamic Array: Resizing



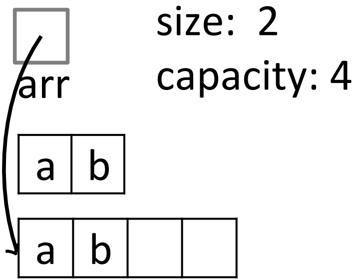
size: 2

capacity: 2

`add(c)`

Cannot add c: need to resize

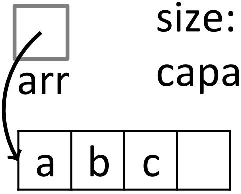
Dynamic Array: Resizing



add(c)

Resize array: copy old data

Dynamic Array: Resizing

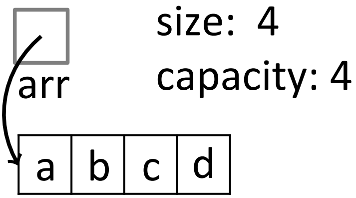


size: 3

capacity: 4

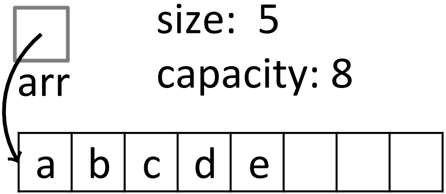
add(c)

Dynamic Array: Resizing



add(d)

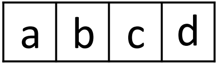
Dynamic Array: Resizing




```
add(e)
```

Dynamic Arrays: Common Operations

	Add	Remove
Beginning	O(n)	O(n)
End	O(n)	O(1)
Middle	O(n)	O(n)



add(e)

You always can have a situation when you need to resize and copy in O(n) time

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.

Amortized analysis

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.

- In dynamic arrays we only resize every so often.
- Many O(1) operations are followed by an O(n) operation.
- What is the total cost of inserting many elements?

Definition

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

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

Dynamic arrays: cost of *n* calls to *add*

Let c_i = cost of ith add.

If we choose the strategy to double the size of the array on resizing, then:

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

Dynamic arrays: cost of *n* calls to *add*

Let c_i = cost of ith add.

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

The total cost of performing n insertions:

$$\sum_{i=1}^{n} c_i = n + \sum_{j=1}^{\log n} 2^j = 2n$$

And the amortized cost per insertion: O(n)/n = O(1)

Alternatives to doubling

- We could use some different growth factor (1.5, 2.5, 3 etc.).
- Could we use a constant amount?

Adding a constant amount while resizing

Let's expand by 10 each time, then:

Let c_i = cost of ith add.

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

$$\sum_{i=1}^{n} c_i = n + \sum_{j=1}^{n/10} 10j = n + \frac{0.10n(n-1)}{2} = O(n^2)$$

And the amortized cost per insertion: $O(n^2)/n = O(n)$

Dynamic arrays: common implementations

- **C++**: vector
- **Java**: ArrayList
- **Python**: list (some indirection)

Digression: Python arrays

- An array in Python is called *a list*
- Underneath there is a C-array
- Arrays in Python do not store contiguous data, but store contiguous **pointers** to data
- Arrays in Python are implemented as dynamic (resizable) arrays of pointers (addresses)

https://www.laurentluce.com/posts/python-list-implementation/

Summary

- We learned how to calculate amortized cost of an operation in the context of a sequence of operations.
- We used a brute-force summation. This is called the *Aggregate Method* of amortized analysis
- There are other methods for more complex cases:
  - Banker's method (tokens)
  - Physicist's method (potential function, Φ)

Amortized analysis is a useful tool, because we can adjust the implementation of a data structure based on the aggregated time