Data structures: motivation

- The choice of a suitable data structure can make all the difference between an efficient and a failing program
- The input and output of any algorithm is stored inside a data structure
- Data structures organize data for quick and efficient access

Examples of data structures

- Simple: ***lists***, ***stacks*** and ***queues***
- More intricate - but still very useful: ***heaps***, ***search trees***, ***hash tables***, ***Bloom filters***, ***union-find*** …

- Why so many?

  Because different data structures support different sets of operations and are good for different types of tasks

Know what exists and what it is good for

- We will discuss the pros and cons of each data structure for a particular task
- The fewer operations the data structure supports - the faster the operations will be

Think about the operations that you need a data structure to support

Choose the best data structure - the one that supports only required operations, and not more.

4 levels of Data Structure Proficiency

- Level 0: **ignorance**
- Level 1: **cocktail party awareness**
- Level 2: **solid literacy**: know which data structures are appropriate for which types of tasks and comfortable using them

- Level 3: **hardcore** programmers and computer scientists: understand the internals of existing and implement new data structures

Basic Data Structures: Arrays and Linked Lists

Lecture 02.01 by Marina Barsky

ARRAYS

```
long arr[] = new long[5];
```

Java

```
long arr[5];
```

C/C++

```
arr = [None] * 5
```

Python

1	5	17	3	25

1D

1	5	17	3	25
8	2	36	5	3

2D

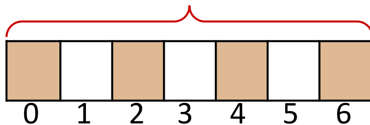
Definition

Array:

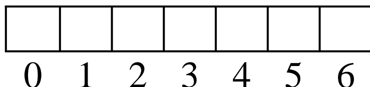
Contiguous area of memory containing **equal-size** elements indexed by contiguous integers.

The maximum number of elements that can fit into the allocated memory is called a *capacity* of the array.

The number of elements currently in the array is called a *size* of the array.



What's Special About Arrays?



Constant-time access to any element by index *i*.

Computed as:

array_addr + elem_size × (*i*)

Multi-Dimensional Arrays

int arr [3][6];

(0,0)					
			(2,3)		

$$\text{array_addr} + \text{elem_size} \times (2 \times 6 + 3)$$

(0, 0)
(0, 1)
(0, 2)
(0, 3)
(0, 4)
(0, 5)
(1, 0)
.

Arrays: Time for Common Operations

	Add	Remove
Beginning		
End		
Middle		

5	8	3	12			

`size=4`

Arrays: Time for Common Operations

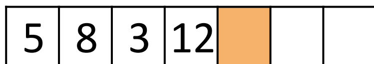
	Add	Remove
Beginning		
End	O(1)	
Middle		

5	8	3	12	4		

```
A[4] = 4
size = 5
```

Arrays: Time for Common Operations

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



Remove A[4]
size=4

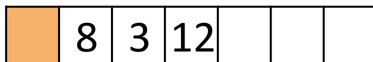
Arrays: Time for Common Operations

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

5	8	3	12			

Arrays: Time for Common Operations

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



Arrays: Time for Common Operations

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

8		3	12			

Arrays: Time for Common Operations

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

8	3		12			

Arrays: Time for Common Operations

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

8	3	12				

size=3

Arrays: Time for Common Operations

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

size=3

Arrays: Time for Common Operations

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

size=3

Summary: Arrays

- Array: contiguous area of memory consisting of equal-size elements indexed by contiguous integers
- Constant-time access to any element by location (index)
- Constant time to add/remove at the end
- Linear time to add/remove at an arbitrary location