CSE 250
Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu
208 Capen Hall

Lec 14: Midterm #1 Review

Midterm Procedure

- Exam is during normal class time. Same time, same place.
- Seating is assigned randomly
  - Wait outside the room until instructed to enter
  - Immediately place all bags/electronics at the front of the room
- At your seat you should have:
  - Writing utensil
  - UB ID card
  - One 8.5x11 cheatsheet (front and back) if desired
  - Summation/Log rules will be provided

Content Overview

	Analysis Tools/Techniques	ADTs	Data Structures
Week 2/3	Asymptotic Analysis, (Unqualified) Runtime Bounds		
Week 3		Sequence	Array, LinkedList
Week 4	Amortized Runtime	List	ArrayList, LinkedList
Week 5	Induction, Expected Runtime	Stack/Queue	ArrayList, LinkedList

Analysis Tools and Techniques

Recap of Runtime Complexity

Big-Θ — Tight Bound

- Growth functions are in the **same** complexity class
- If f(n) ∈ **Θ**(g(n)) then an algorithm taking f(n) steps is as "exactly" as fast as one that takes g(n) steps.

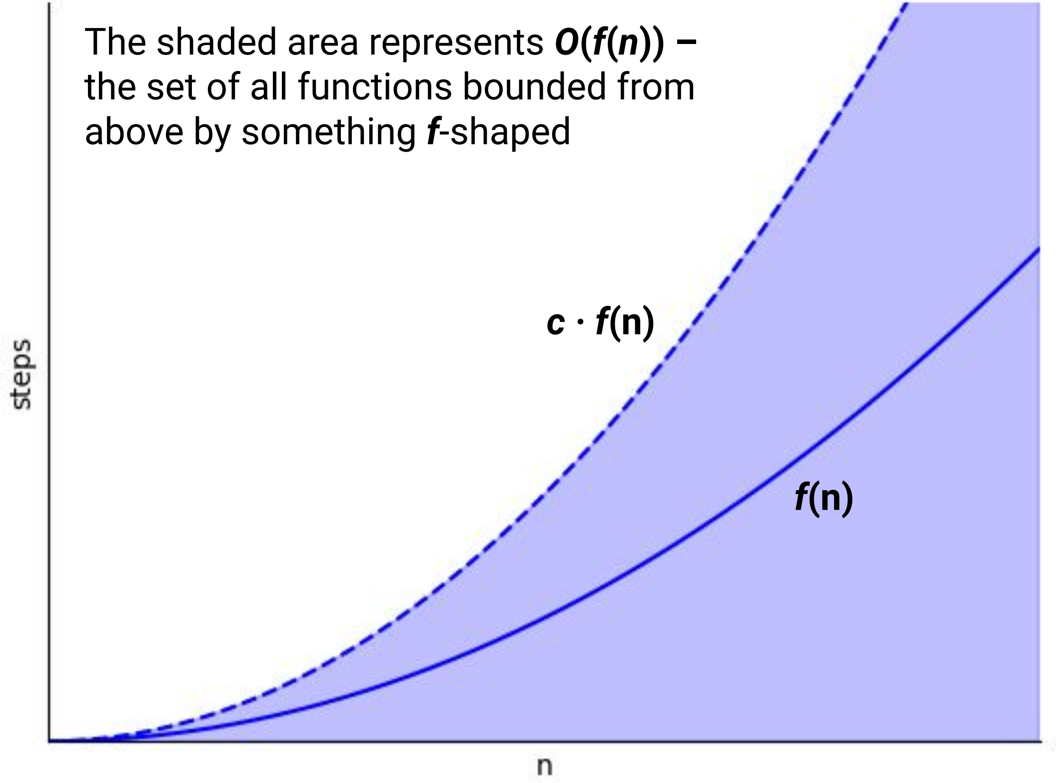
Big-O — Upper Bound

- Growth functions in the **same or smaller** complexity class.
- If f(n) ∈ O(g(n)), then an algorithm that takes f(n) steps is *at least as fast* as one taking g(n) (but it may be even faster).

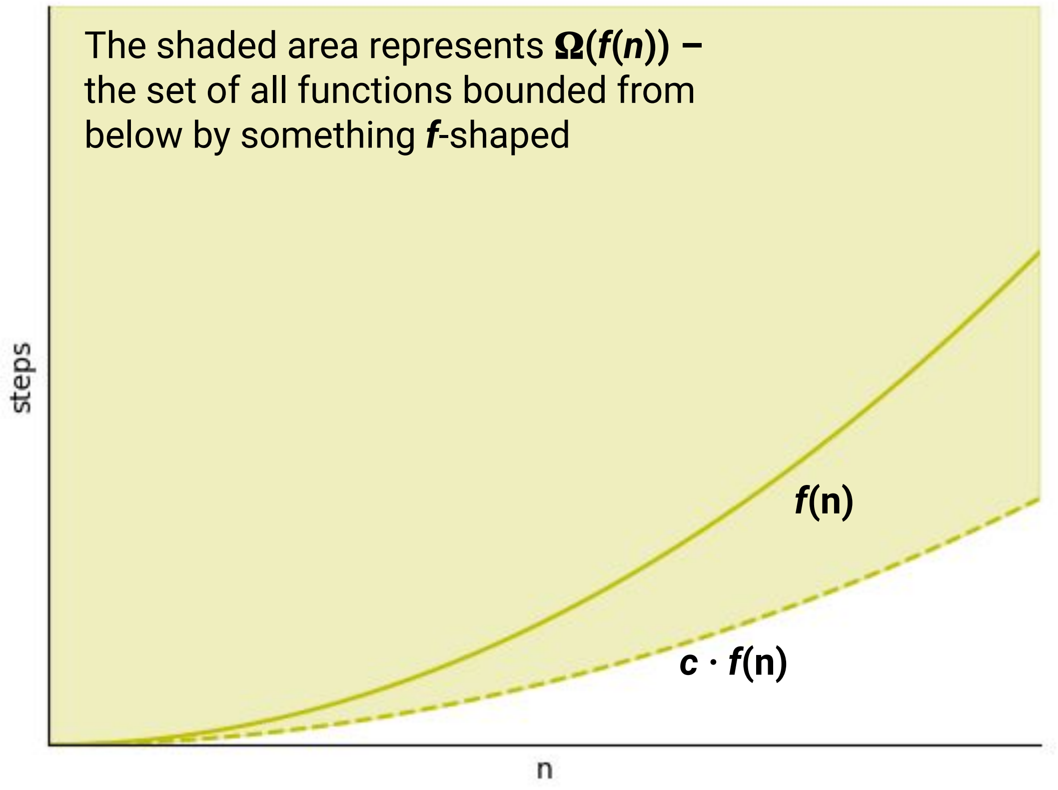
Big-Ω — Lower Bound

- Growth functions in the **same or bigger** complexity class
- If f(n) ∈ **Ω**(g(n)), then an algorithm that takes f(n) steps is *at least as slow* as one that takes g(n) steps (but it may be even slower)

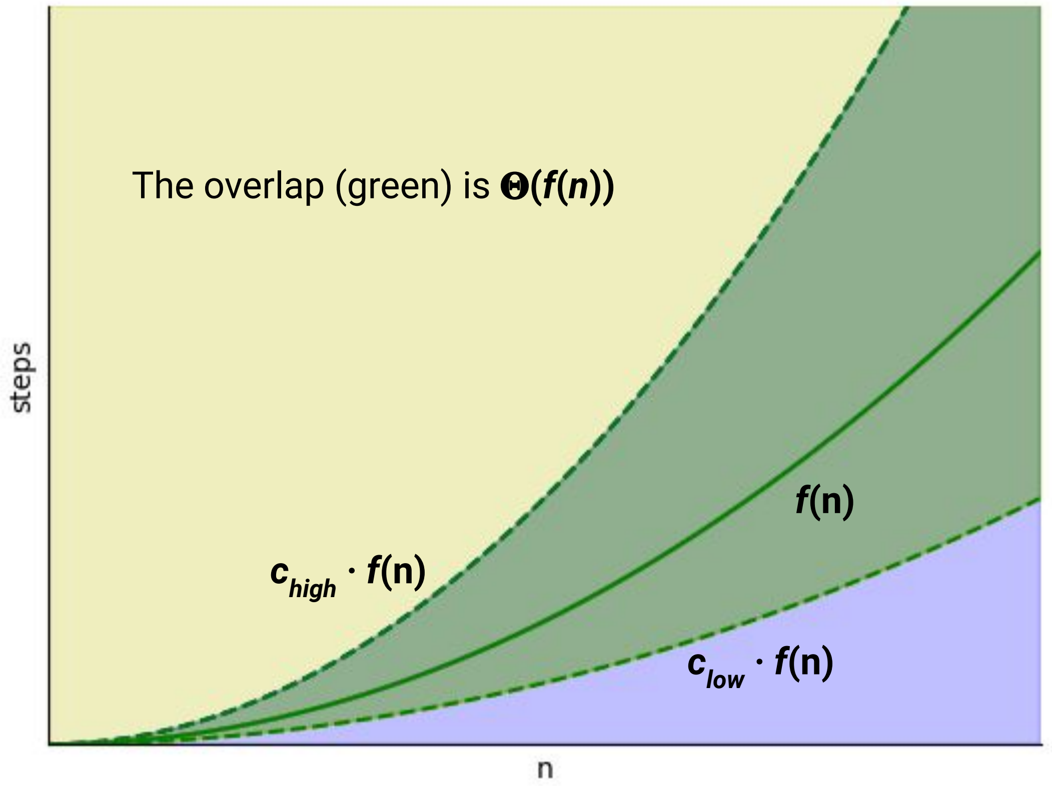
Bounded from Above: Big O



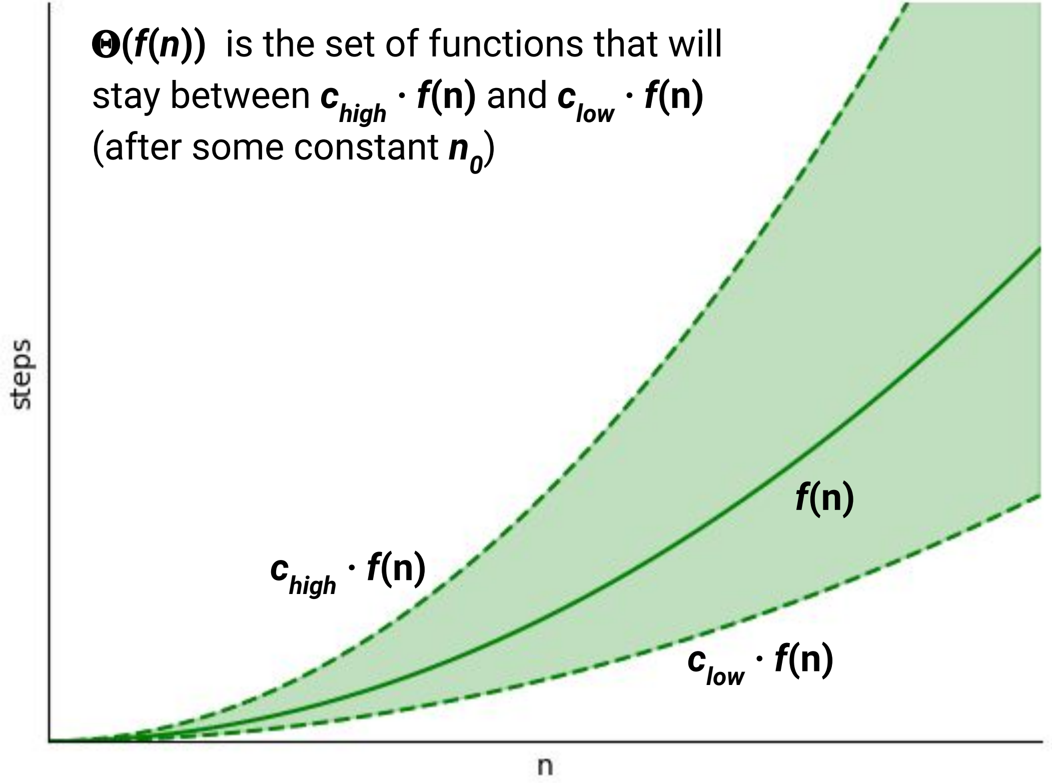
Bounded from Below: Big Ω



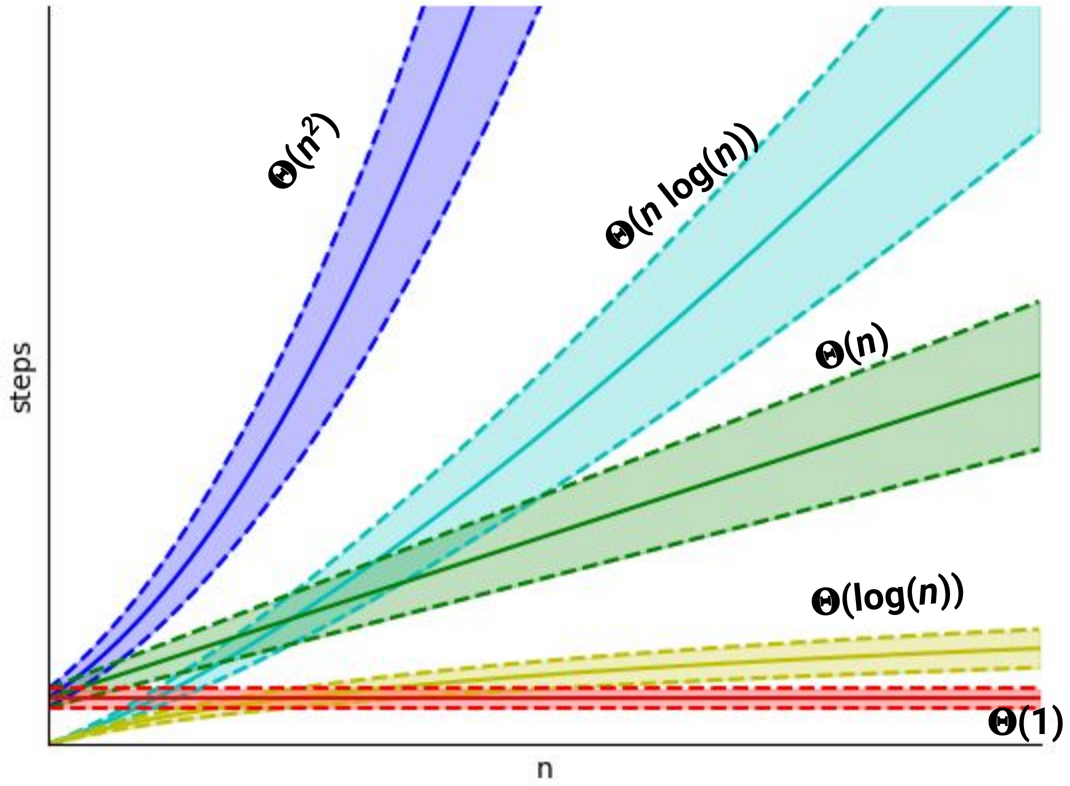
Complexity Class: Big Θ



Complexity Class: Big Θ



Complexity Class Ranking



$\Theta(1) < \Theta(\log(n)) < \Theta(n) < \Theta(n \log(n)) < \Theta(n^2) < \Theta(n^3) < \Theta(2^n)$

Common Runtimes (in order of complexity)

Constant Time:	**$\Theta(1)$**
Logarithmic Time:	**$\Theta(\log(n))$**
Linear Time:	**$\Theta(n)$**
Quadratic Time:	**$\Theta(n^2)$**
Polynomial Time:	**$\Theta(n^k)$ for some $k > 0$**
Exponential Time:	**$\Theta(c^n)$ (for some $c \geq 1$)**

Formal Definitions

$\mathbf{f(n)} \in \boldsymbol{O}\mathbf{(g(n))}$ iff exists some constants $\mathbf{c}$, $\mathbf{n_0}$ s.t.

$\mathbf{f(n) \leq c * g(n)}$ for all $\mathbf{n > n_0}$

$\mathbf{f(n)} \in \boldsymbol{\Omega}\mathbf{(g(n))}$ iff exists some constants $\mathbf{c}$, $\mathbf{n_0}$ s.t.

$\mathbf{f(n) \geq c * g(n)}$ for all $\mathbf{n > n_0}$

$\mathbf{f(n)} \in \boldsymbol{\Theta}\mathbf{(g(n))}$ iff $\mathbf{f(n)} \in \boldsymbol{O}\mathbf{(g(n))}$ and $\mathbf{f(n)} \in \boldsymbol{\Omega}\mathbf{(g(n))}$

Shortcut

What complexity class do each of the following belong to:

$\boldsymbol{f(n) = 4n + n^2 \in \Theta(n^2)}$

$\boldsymbol{g(n) = 2^n + 4n \in \Theta(2^n)}$

$\boldsymbol{h(n) = 100\, n \log(n) + 73n \in \Theta(n \log(n))}$

Shortcut: Just consider the complexity of the most dominant term

Multi-class Functions

$$T(n) = \begin{cases} n^2 & \text{if } n \text{ is even} \\ n & \text{if } n \text{ is odd} \end{cases}$$

It is not bounded from above by $\boldsymbol{n}$, therefore it cannot be in $\boldsymbol{\Theta(n)}$

It is not bounded from below by $\boldsymbol{n^2}$, therefore it cannot be in $\boldsymbol{\Theta(n^2)}$

What is the tight upper bound of this function? $\boldsymbol{T(n) \in O(n^2)}$

What is the tight lower bound of this function? $\boldsymbol{T(n) \in \Omega(n)}$

What is the complexity class of this function? It does not have one!

Amortized Runtime

If n calls to a function take $\Theta(f(n))$...

We say the **Amortized Runtime** is $\Theta(f(n)\ /\ n)$

The **amortized runtime** of `add` on an `ArrayList` is: $\Theta(n/n) = \Theta(1)$

The **unqualified runtime** of `add` on an `ArrayList` is: $O(n)$

Expected Runtime

If our algorithm involves some sort of random process, we can still analyze the runtime as a growth function **T(*n*)**...

But we can also analyze the <u>**expected runtime**</u>, **E[T(*n*)]**

Example: $T_{quicksort}(n) \in O(n^2)$ and $E[T_{quicksort}(n)] \in O(n \log(n))$

What guarantees do you get?

If $f(n)$ is a Tight Bound

The algorithm always runs in $cf(n)$ steps

← Unqualified runtime

If $f(n)$ is a Worst-Case Bound

The algorithm always runs in at most $cf(n)$

If $f(n)$ is an Amortized Worst-Case Bound

n invocations of the algorithm **always** run in $cnf(n)$ steps

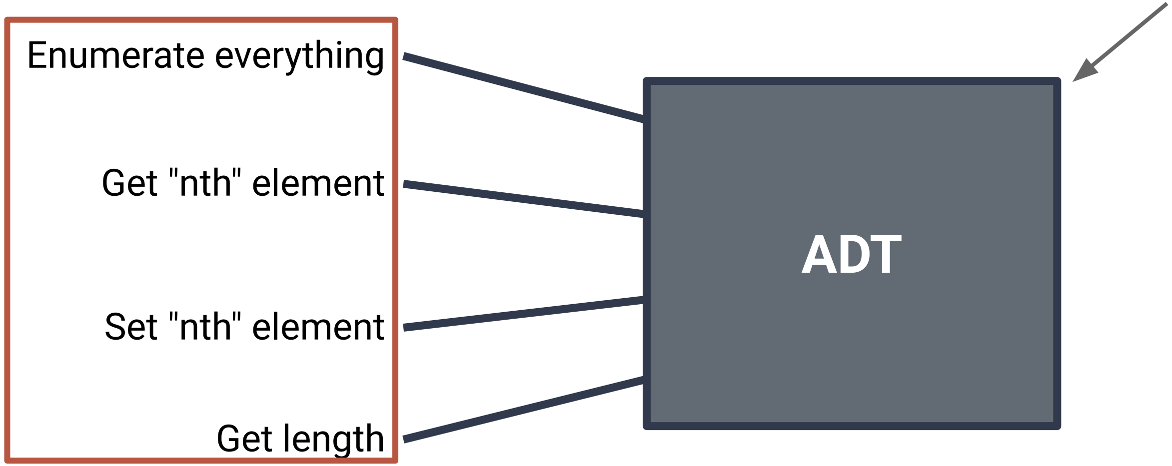
If $f(n)$ is an Average Bound

...we don't have any guarantees

ADTs and Data Structures

Abstract Data Types (ADTs)

The specification of **what** a data structure can do



What's in the box? …we don't know, and in some sense…we don't care

Usage is governed by **what** we can do, not **how** it is done

Abstract Data Type vs Data Structure

ADT

The interface to a data structure

*Defines **what** the data structure can do*

Many data structures can implement the same ADT

Data Structure

The implementation of one (or more) ADTs

*Defines **how** the different tasks are carried out*

Different data structures will excel at different tasks

Abstract Data Type vs Data Structure

ADT

The interface to

*Defines **what** the*

can

Many data st

implement th

Data Structure

tation of one (or

) ADTs

e different tasks

ried out

uctures will excel

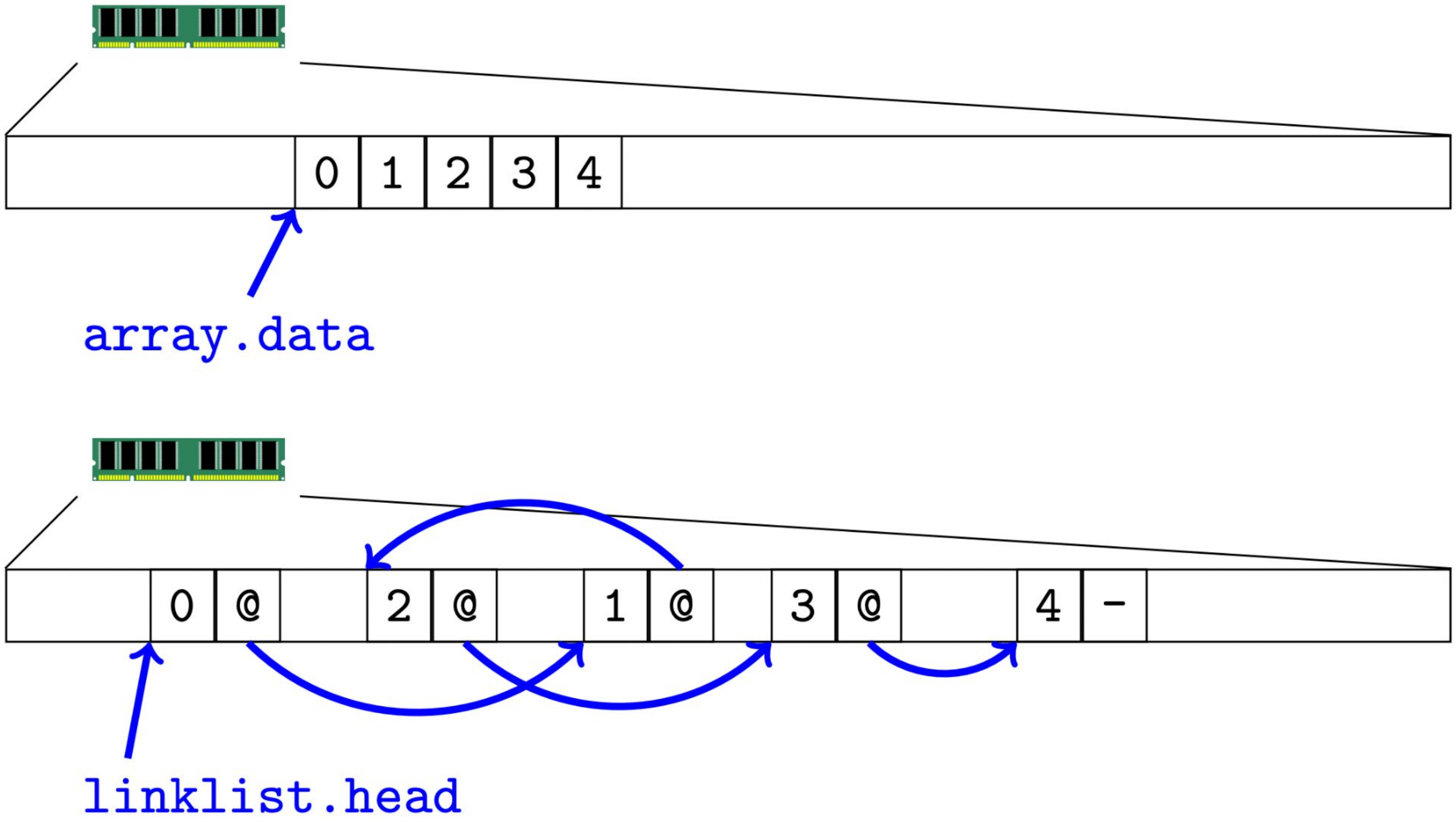
at different tasks

Think about the Linked List we implemented for PA1.

The internal structure and the mental model of our sequence are very different.

The Sequence ADT

```
public interface Sequence<E> {
  public E get(idx: Int);
  public void set(idx: Int, E value);
  public int size();
  public Iterator<E> iterator();
}
```



Arrays and Linked Lists in Memory

The List ADT

```
public interface List<E>
    extends Sequence<E> { // Everything a sequence has, and...
  /** Extend the sequence with a new element at the end */
  public void add(E value);

  /** Extend the sequence by inserting a new element */
  public void add(int idx, E value);

  /** Remove the element at a given index */
  public void remove(int idx);
}
```

Runtime Summary

	ArrayList	Linked List (by index)	Linked List (by reference)
`get(...)`	$\Theta(1)$	$\Theta(\text{idx})$ or $O(n)$	$\Theta(1)$
`set(...)`	$\Theta(1)$	$\Theta(\text{idx})$ or $O(n)$	$\Theta(1)$
`size()`	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`add(...)`	$O(n)$, Amortized $\Theta(1)$	$\Theta(\text{idx})$ or $O(n)$	$\Theta(1)$
`remove(...)`	$O(n)$	$\Theta(\text{idx})$ or $O(n)$	$\Theta(1)$

Runtime Summary

	ArrayList	Linked List (by index)	Linked List (by reference)
`get(...)`	$\Theta(1)$	$\Theta(\text{idx})$ or $O(n)$	$\Theta(1)$
`set(...)`			
`size()`			
`add(...)`	$O(n)$, A		
`remove(...)`	$O(n)$	$\Theta(\text{idx})$ or $O(n)$	$\Theta(1)$

Also consider how we can search by value… ie searching for a value in our SortedList from PA1, searching an unsorted Array vs searching a sorted Array, etc…

Stacks

Represents a stack of objects on top of one another

```java
public class Stack<E> {

  public void push(E value); // Add value to the "top" of the stack

  public E pop();    // Remove and return the top of the stack

  public E peek();  // Return the top of the stack

}
```

Queues

Outside of the US, "queueing" is lining up, ie at Starbucks

```
public class Queue<E> {

  public void add(E value); // Add value to the "back" of the queue

  public E remove();    // Remove and return the front of the queue

  public E peek();  // Return the front of the queue

}
```

Recap

Stacks: Last In First Out (LIFO)

- Push (put item on top of the stack) $\Theta(1)$ (or amortized $O(1)$)
- Pop (take item off top of stack) $\Theta(1)$
- Peek (peek at top of stack) $\Theta(1)$

Queues: First in First Out (FIFO)

- Enqueue (put item on the end of the queue) $\Theta(1)$ (or amortized $O(1)$)
- Dequeue (take item off the front of the queue) $\Theta(1)$
- Peek (peek at the item in the front of the queue) $\Theta(1)$