CS 1181
Week Eight

Reese Hatfield

Lists and Maps

Reese Hatfield

Array and Lists

- The (two) types of lists we've seen
  - Arrays
  - ArrayLists

- What are the differences
- Why use one over the other

Array and Lists

- Odds are:
  - Learned Arrays
  - Learned ArrayLists were easier
  - You *always* use ArrayList now

- But why did make this abstraction in the first place

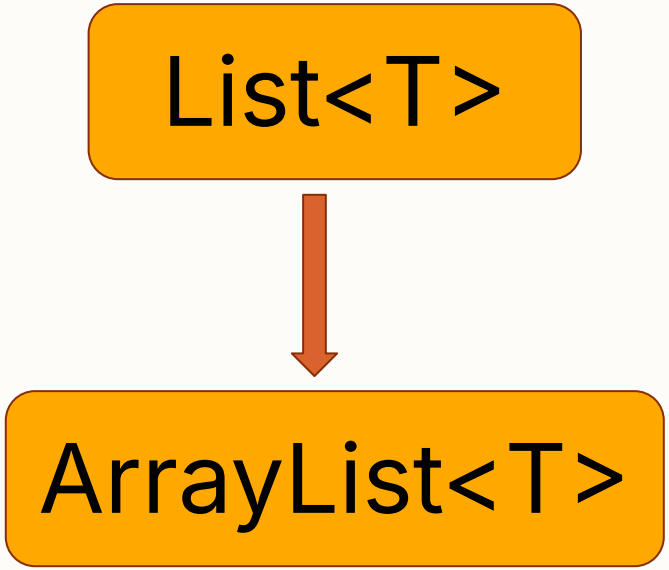
Array and Lists

- What was annoying about arrays?
  - Need index to add a value
  - Fixed size
  - Need to know the size at creation
- ArrayLists are “dynamic”
  - .add()
  - .size()

Array and Lists

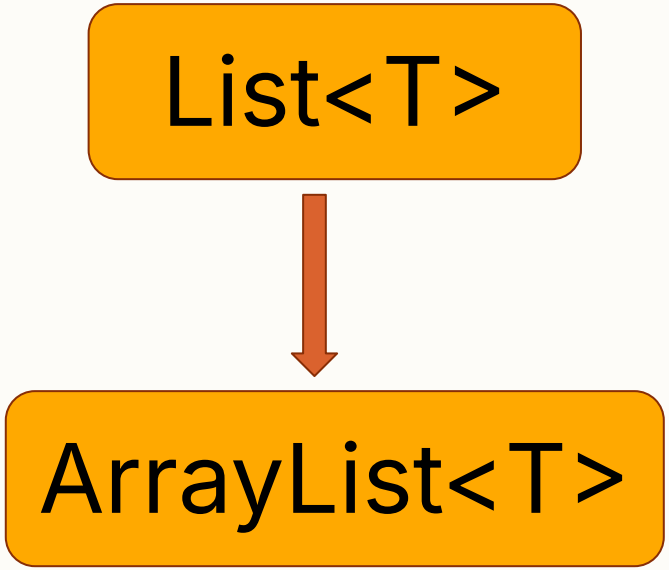
- ArrayList inherits from AbstractList

- Why differentiate these?
- ArrayList specific type of List

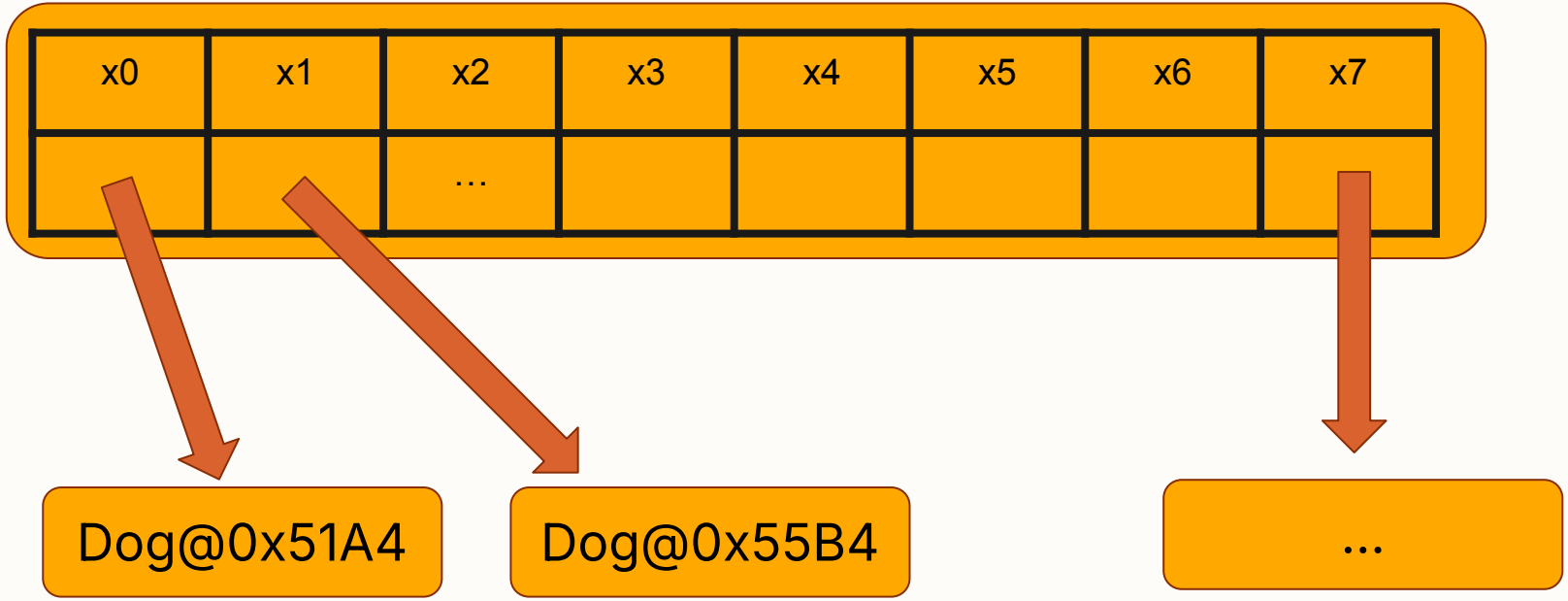


Array and Lists

- AL treats the data as an array
  - get(index)
- What does this mean in terms of memory



Array and Lists



Array and Lists

- Contiguous memory structure
  - Right next to one another

- This allows for:
  - Quick lookup
  - Jump to position immediately

Array and Lists

- ArrayList<T> L = new ArrayList<>();

Memory Location	x0	x1	x2	x3	x4	x5	x6
Data	Value1	Value2	…				

- This gets me a “pointer” to the 0th position
  - How would I get to the 4th position?

Array and Lists

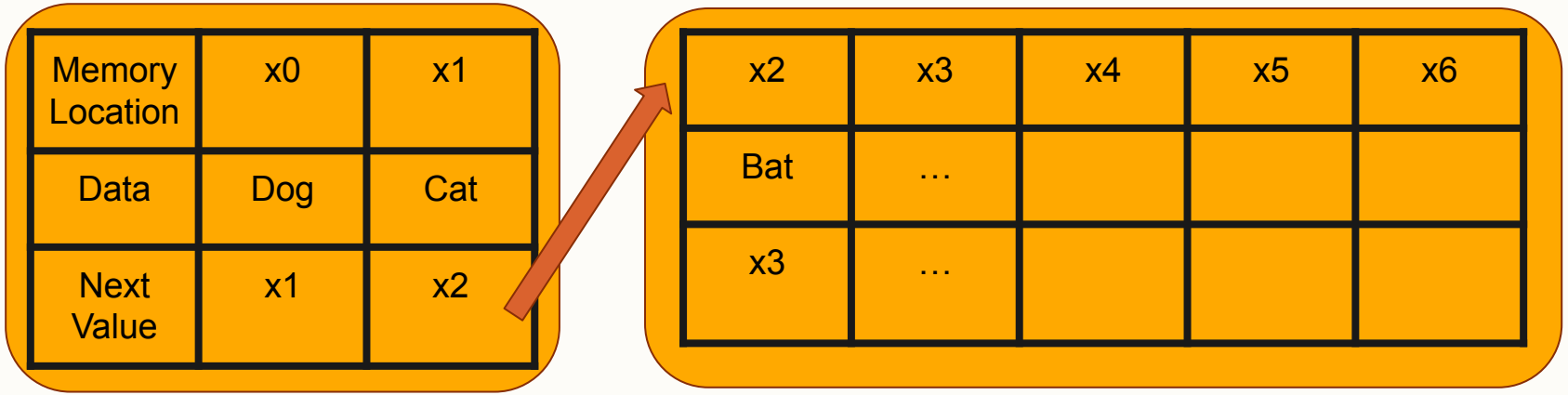
- How else could I organize my list?
- Especially if memory is just a giant array?
  - Right next door.
  - Somewhere else?

Array and Lists

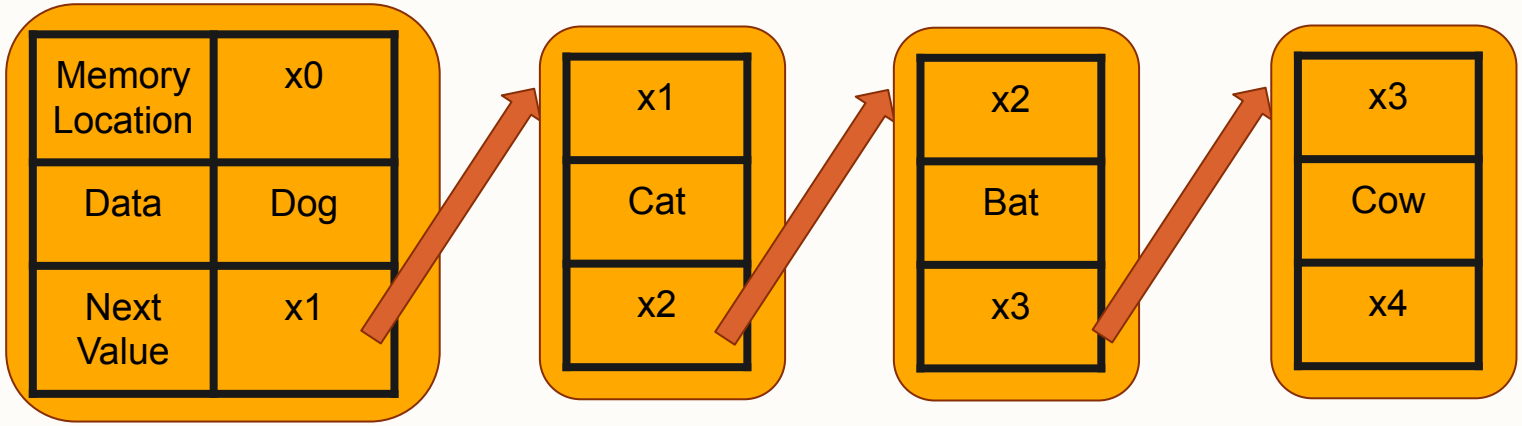
- Let’s aggregate a single piece of data
  - Data Value
  - Where the next piece of data is

Memory Location	x0	x1	x2	x3	x4	x5	x6
Data	Dog	Cat	Bat	…			
Next Value	x1	x2	x3	…			

Array and Lists

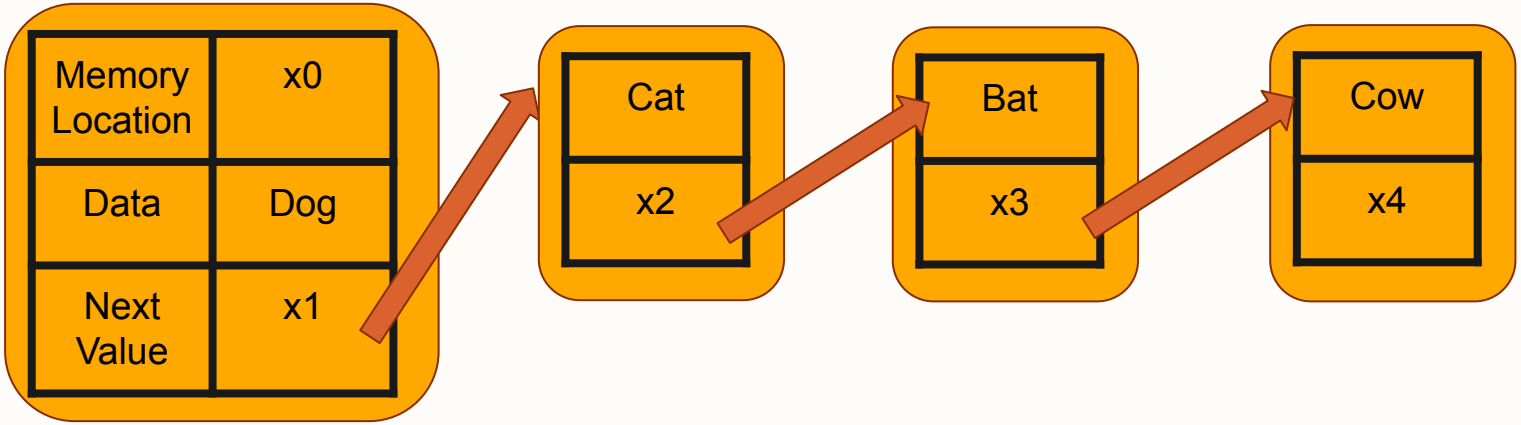


Array and Lists



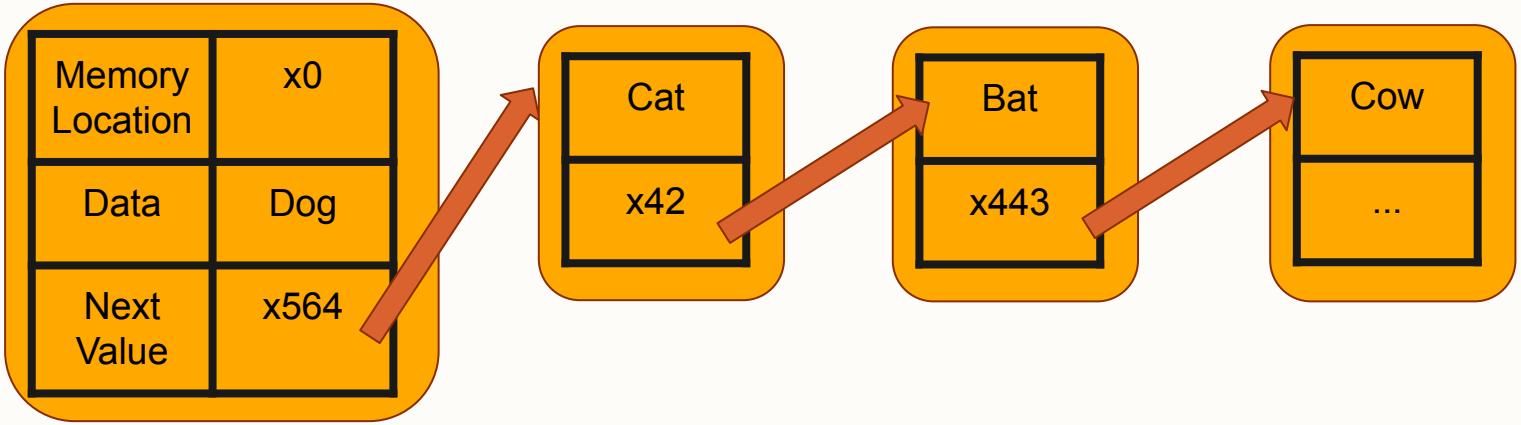
Array and Lists

- Memory locations omitted
- No longer need to be contiguous



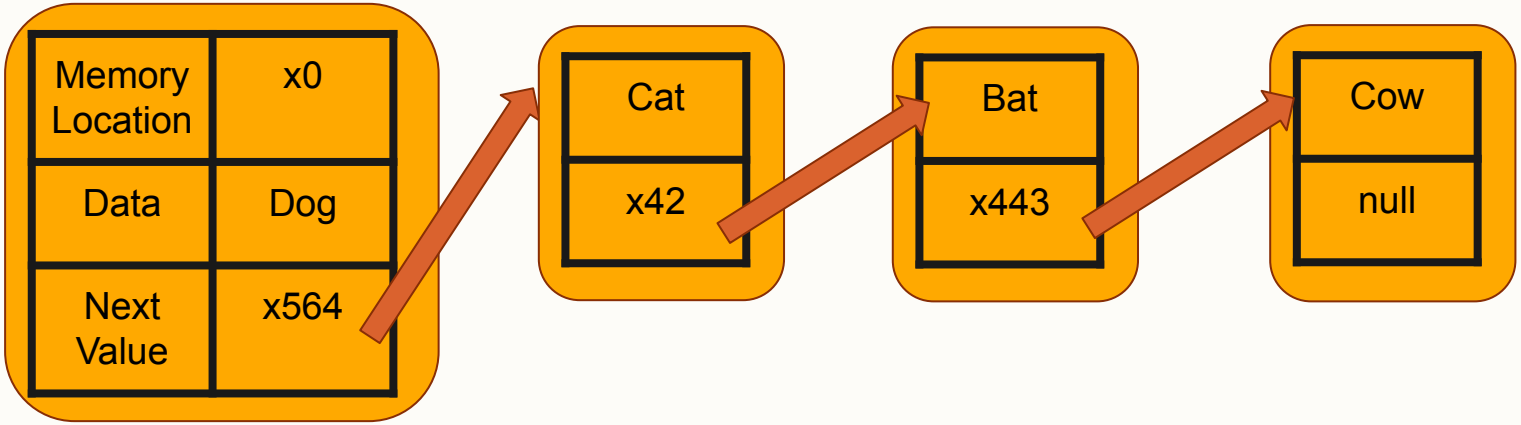
Array and Lists

- Points to random locations
- Where is convenient to allocate



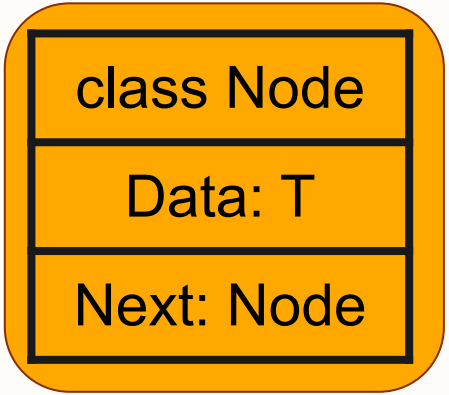
Array and Lists

- Usually “null” terminated
- What does that mean?



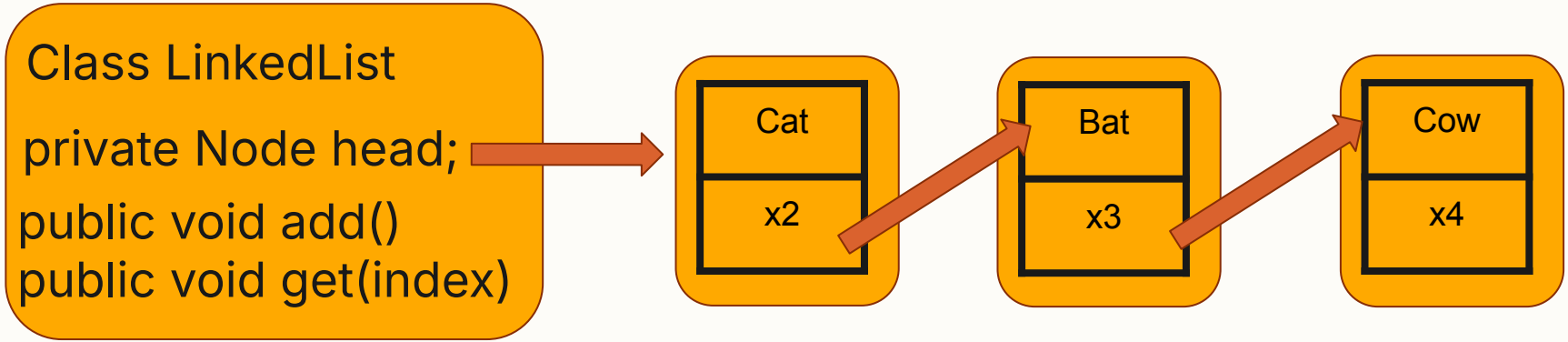
Array and Lists

- Wrap data in a Node class
- Pay attention to the types
  - Generic
  - Node (itself)



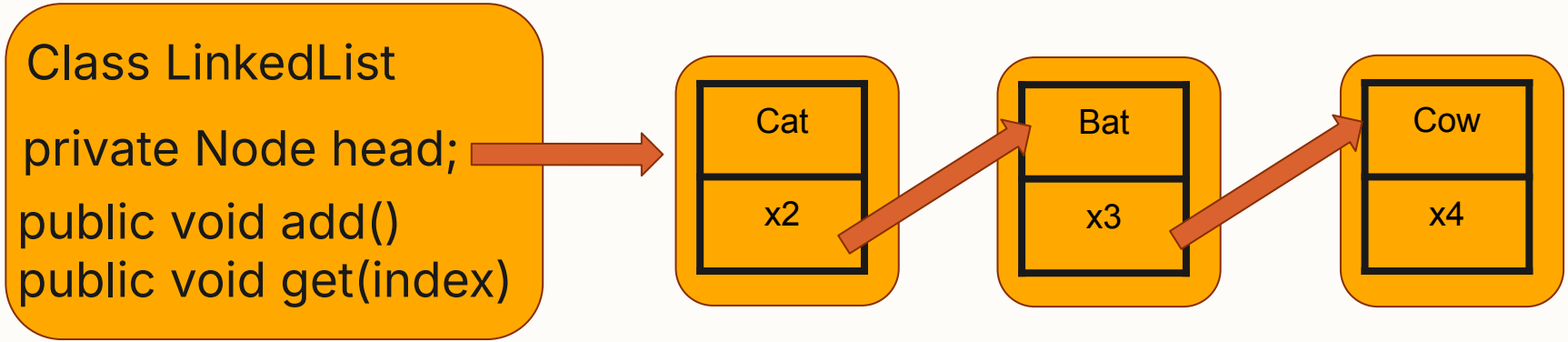
Array and Lists

- Wrap this in a class
  - Only knows about the “head”
- Call it a LinkedList



Array and Lists

- Java provides a LinkedList class
  - Extends from List
- Let’s make our own



Array and Lists

- Why did we do all of that ;-;
  - This is more annoying than arrays to start with
- Let’s look at the time it takes to perform some operations
  - get()
  - add()
  - insert() ?

Array and Lists

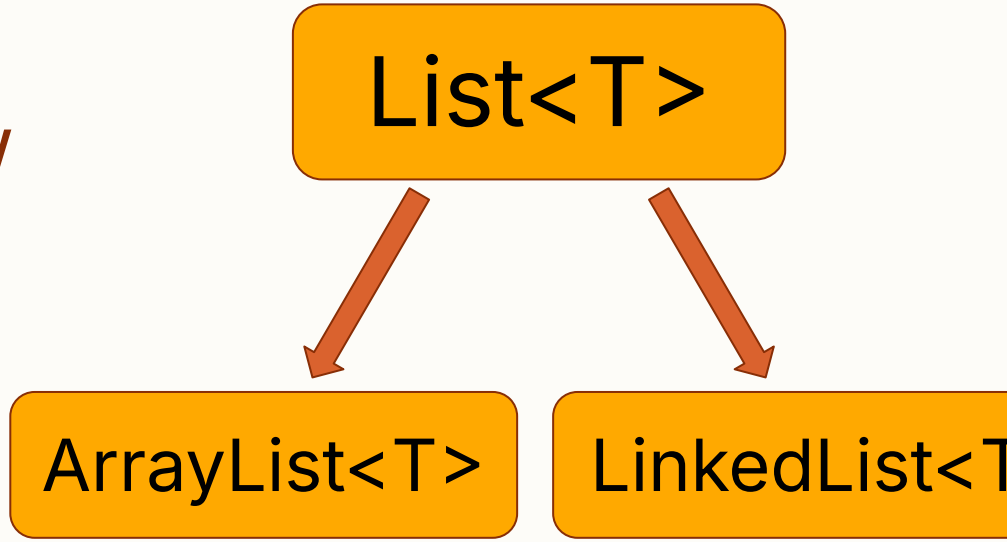
- Let’s program insert() method
- And we can test to see if its faster!
- Next class if time permits
  - I’ll write the benchmark most likely
  - Demo that Thursday

List types

- We've seen two types of lists now

- ArrayList
- LinkedList

- Both implement List<E> interface



Array and Lists

- Trade-offs associated with different operations
- Formalized with Big O notation

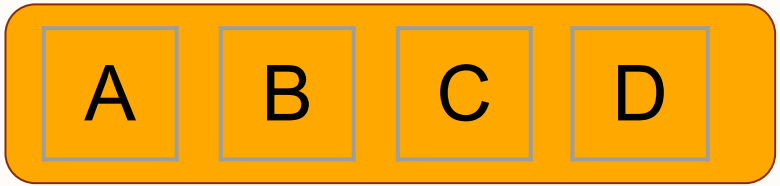
- O(1) → Constant
  - Fast
- O(n) → Linear
  - Slower

Operation	LinkedList	ArrayList
get()	O(n)	O(1)
prepend()	O(1)	O(n)
contains()	O(n)	O(n)

ADTs, Stacks, and Queues

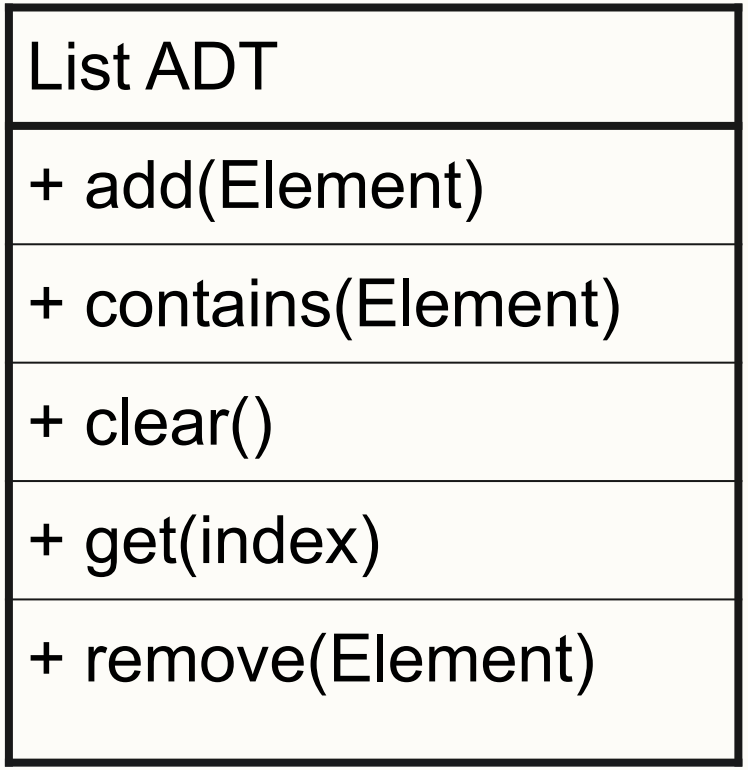
Abstract Data Types (ADTs)

- Everyone has used a List before
- What *actually* makes something a List
- How we can describe the idea of a “List” in more general terms



Abstract Data Types (ADTs)

- Define a series of *ways* to interact with the data
- Tell you *nothing* about how the data is stored



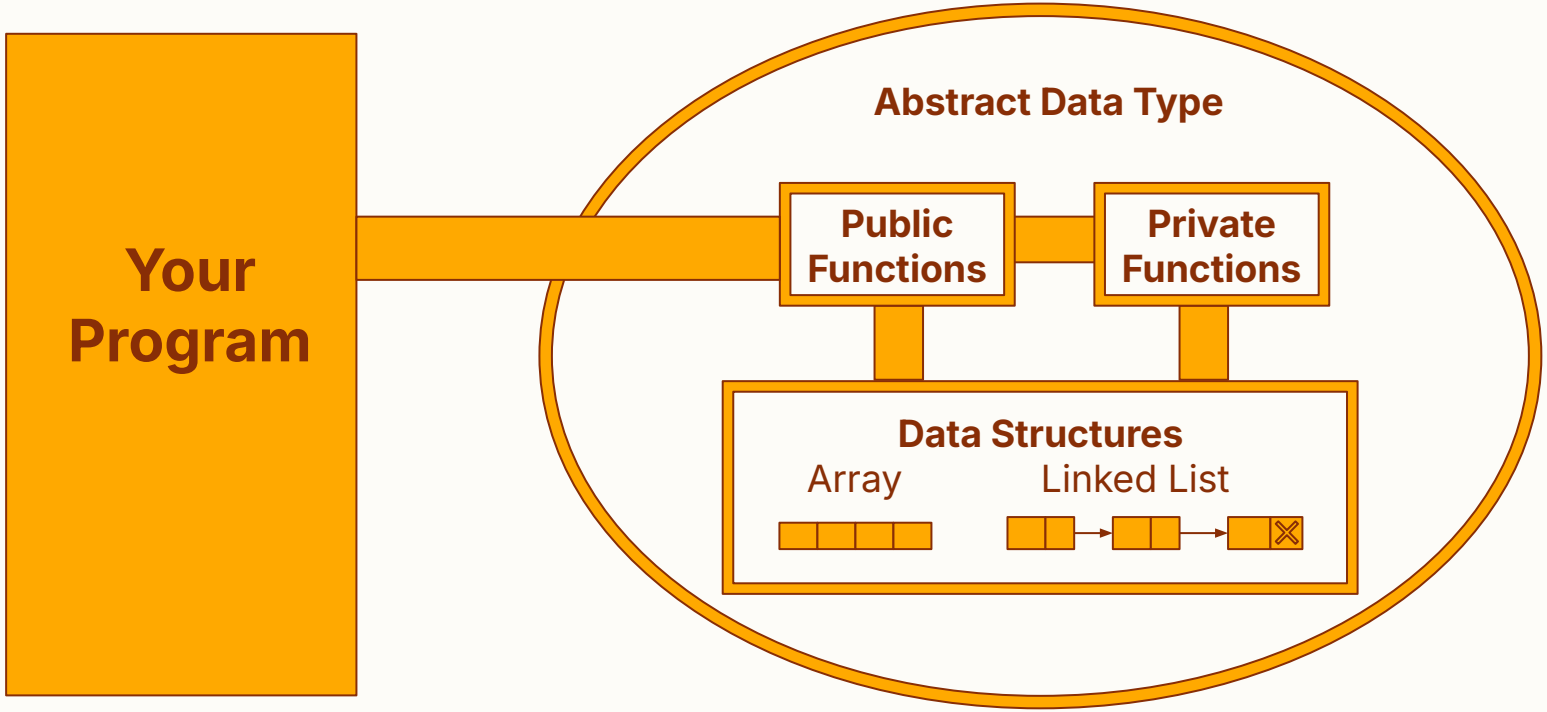
Abstract Data Types (ADTs)

ADTs *do*

- Define operations and methods
- What actions can be performed
- add(), get(), remove(), etc

ADTs do *not*

- Define implementation
- Structure or *type* of underlying data
- Specify performance


Your Program
Abstract Data Type
Public Functions
Private Functions
Data Structures
Array
Linked List

Applied ADTs

- ADTs enable you to focus on solving high-level problems
  - Power in abstraction

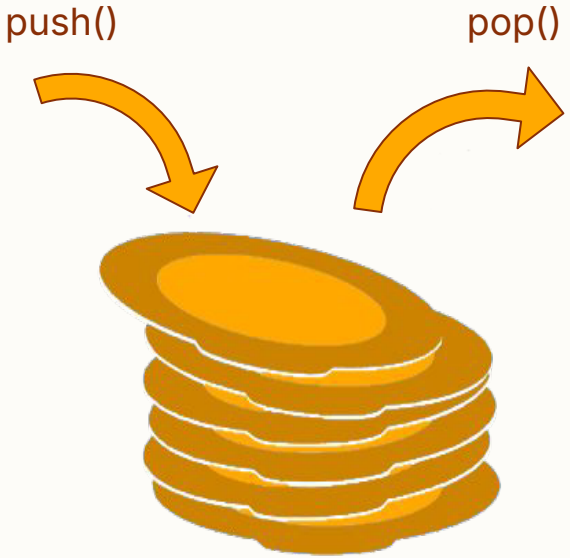
Applied ADTs

- ADTs enable you to focus on solving high-level problems
  - Power in abstraction

- What are some other ADTs we have probably heard of?

Stacks

- Last In → First out
- Only let you modify the thing on top
- Restricts any other operations
- Like a stack of plates



A stack of Plates

Applied Stacks

- Permitted operations
  - push(), pop(), peek()

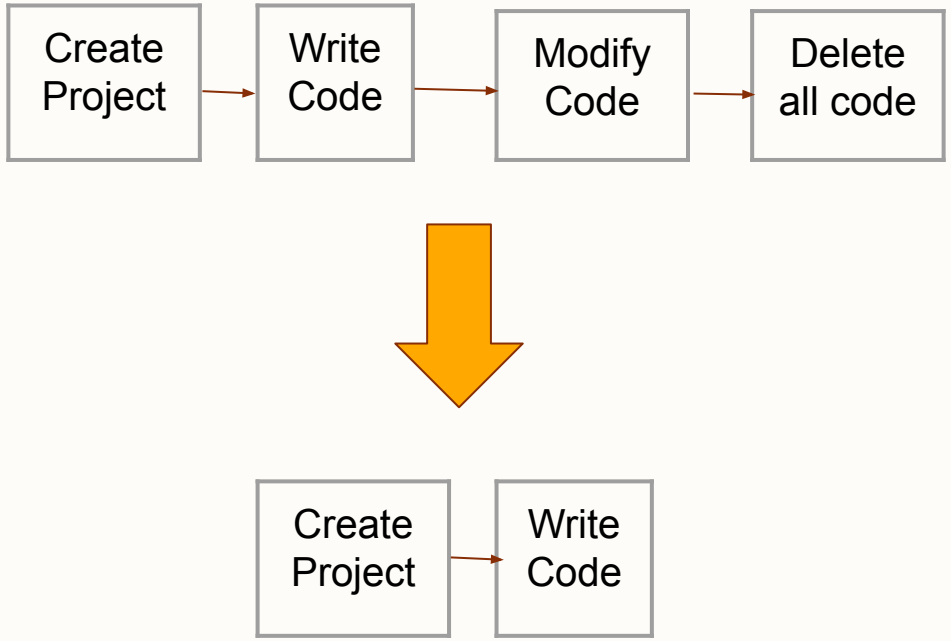
- How should we implement a stack?
  - Linked data structure
  - Contiguous array structure

Stack ADT
+ push(Element)
+ pop(): Element
+ peek(): Element
+ contains(): bool
+ clear()

Problem #1

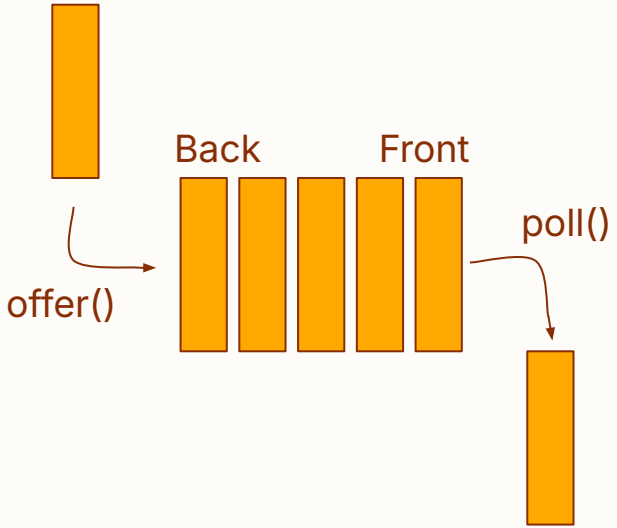
- Input: A sequence of operations.

- Output: The same operations, with the most recent two undone.



Queues

- First In → First out
- Only add to the front
- Remove from the back
- Restricts internal data manipulation
- Like a drive-thru line



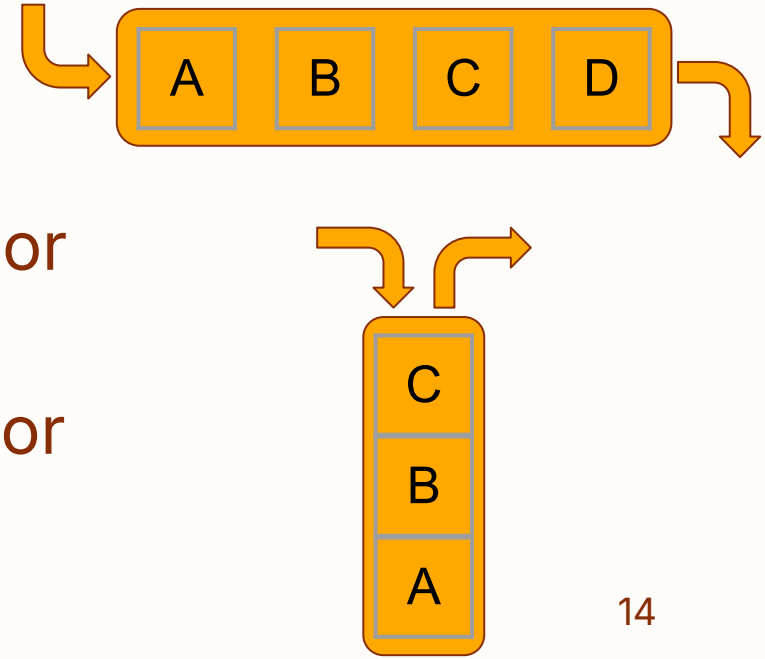
Applied Queues

- Permitted operations
  offer(), poll(), peek(), etc.
- How should we implement a Queue?
  - Linked data structure
  - Contiguous array structure

Queue ADT
+ offer(Element)
+ poll(): Element
+ peek(): Element
+ contains(): bool
+ clear()

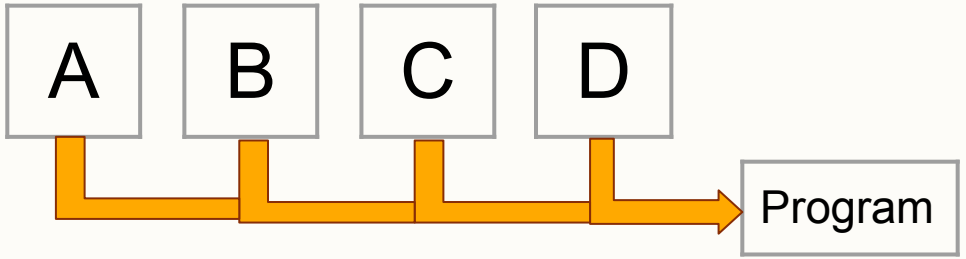
Choosing the right tool

- Different problems call for different ADTs
- Queues excel at modeling scenarios with FIFO behavior
- Stacks excel at modeling scenarios with LIFO behavior
- Let’s do some examples



Problem #2

- Input: A sequence of customers

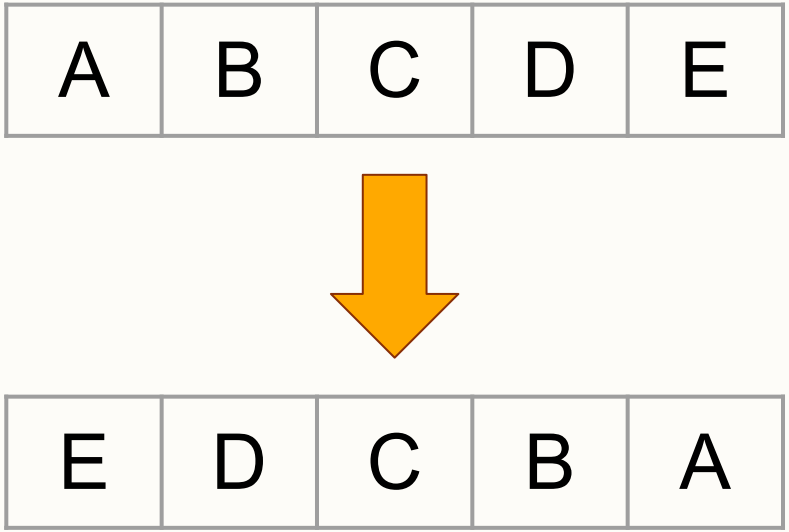


- Output: A log detailing the order of customer arrivals

Arrival Time	1	2	3	4
Customer	A	B	C	D

Problem #3

- Input: a series of elements
- Output: the series of elements in reversed order



Overview

Feature	Stack	Queue
Access Order	LIFO	FIFO
Element availability	Only the top	Only front and back
Common methods	push(), pop(), peek()	offer(), poll(), peek()
Analogy	Stack of Plates	Drive-thru line

Questions?

How would we implement a queue that gives some elements special priority?