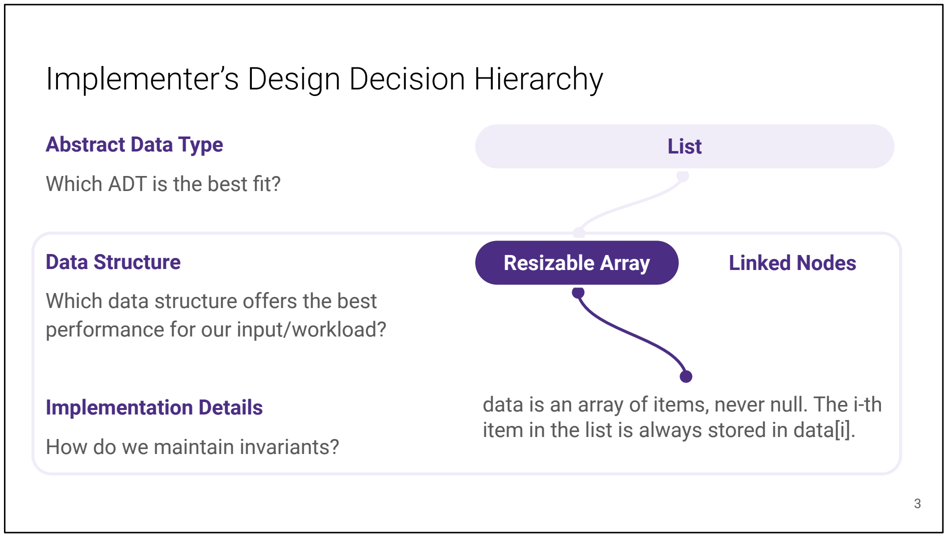
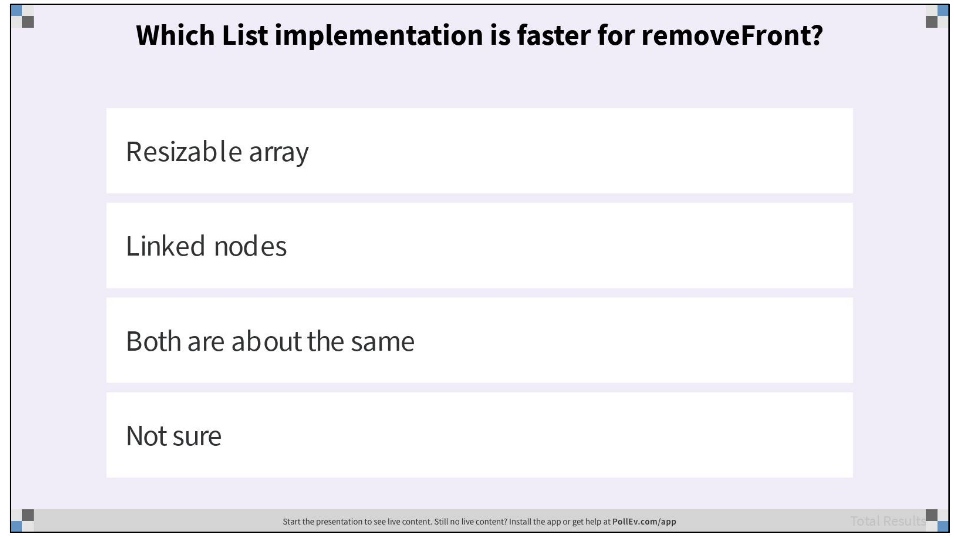




?: How do we determine whether one data structure is faster than another? Does it depend on the implementation details?

?: How do invariants relate to data structures?

Big-O Runtime Analysis

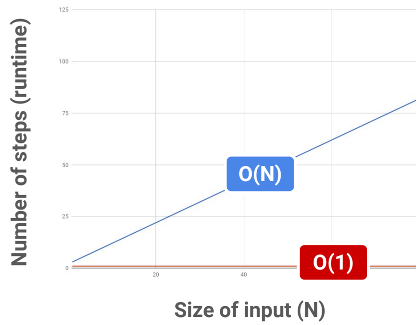
What does it mean for a data structure to be slow or fast?

Big-O runtime analysis: count how many **steps** a program takes to execute an input of size N.

Suppose our list has N items.

A method that takes a **constant** number of steps (e.g. 23) is in **O(1)**.

A method that takes a **linear** number of steps (e.g. 4N + 3) is in **O(N)**.



ArrayList vs. LinkedList

1. Which List implementation should we use to store a list of songs in a playlist?
2. Which List implementation should we use to store the history of a bank customer's transactions?
3. Which List implementation should we use to store the order of students waiting to speak to a TA at a tutoring center?

?: How does **constant** or **linear** relate to analyzing runtime "with respect to big inputs"?

?: What are the big-O runtimes for ArrayList and LinkedList removeFront?

?: Can we say that an ADT is slower or faster than another ADT?

Time needed to access the i-th item from a list of N items.
- ArrayList: O(1)
- LinkedList: O(N)

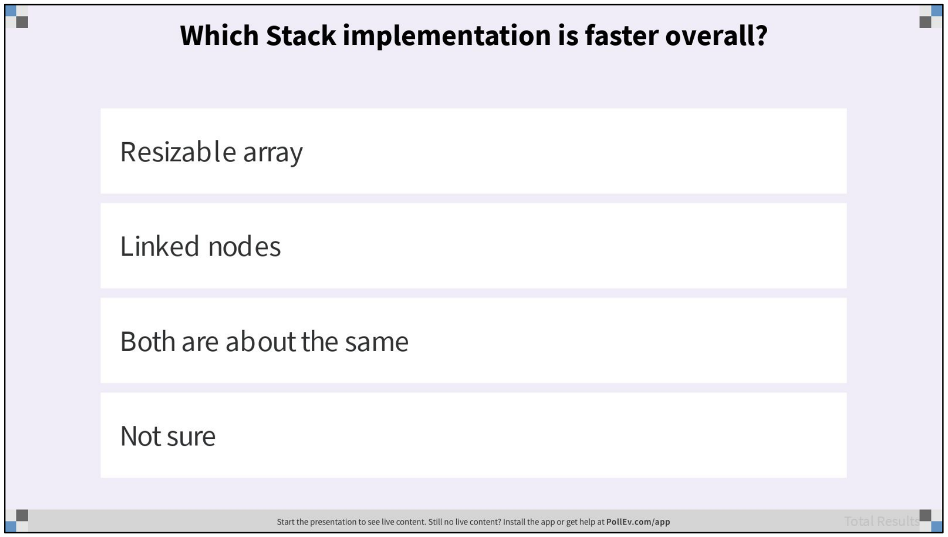
Time needed to insert an item at position i in a list of N items.
- ArrayList: O(N)
- LinkedList: O(N)

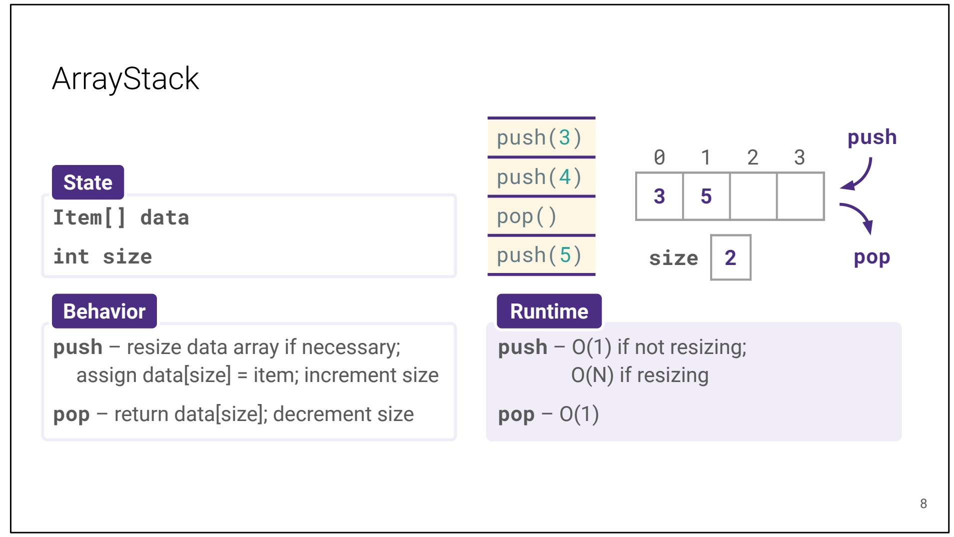
?: Why are these runtimes what they are?

Q1: Which List implementation should we use to store a list of songs in a playlist?

Q2: Which List implementation should we use to store the history of a bank customer's transactions?

Q3: Which List implementation should we use to store the order of students waiting to speak to a TA at a tutoring center?





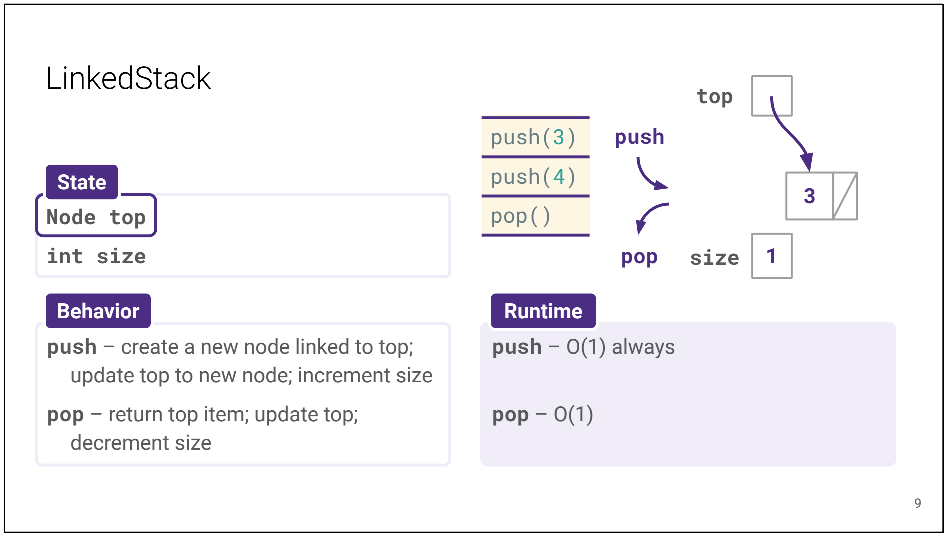
Recall that the Stack ADT specifies two important methods:

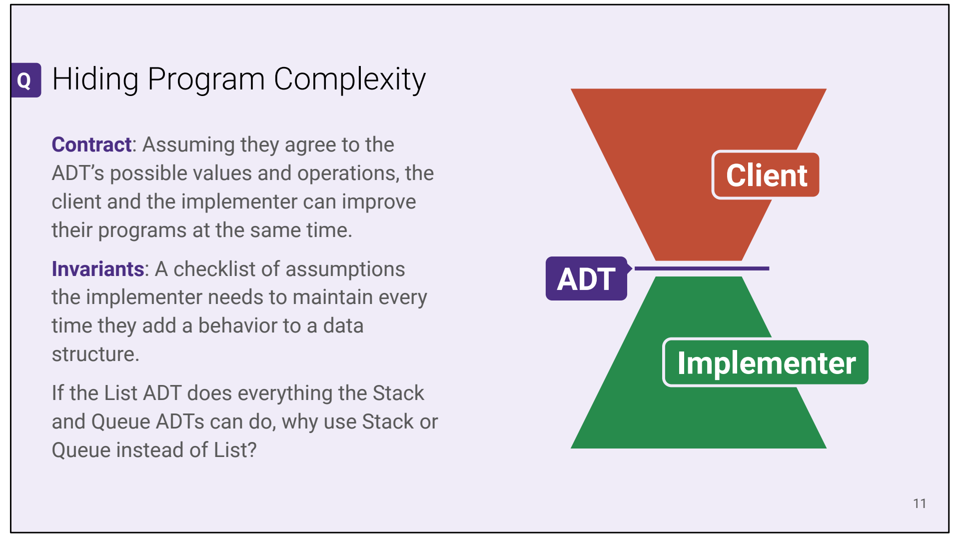
- push(Item item): Puts the item on the top of the stack.
- Item pop(): Removes and returns the top item of the stack.

Assume for the resizable array that we use the addLast and removeLast methods from ArrayList. Assume for linked nodes that we use the addFirst and removeFirst methods from LinkedList, and we have a reference to the front of the LinkedList.

?: How do the Stack ADT methods compare to List ADT methods?

?: How do the implementations for ArrayList methods differ from ArrayStack methods?





?: If the push and pop operations of LinkedStack is always at least as good or better than ArrayStack, would we ever want to use ArrayStack?

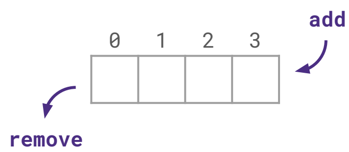
?: How do invariants affect the implementation of ArrayList and ArrayStack?

Q1: If the List ADT does everything the Stack and Queue ADTs can do, why use Stack or Queue instead of List?

ArrayQueue: Design 1

Same design as ArrayStack: borrow ArrayList's addLast and removeFront.

It's basically just an ArrayList.



Q Reconsidering Data Structure Invariants

ArrayQueue (Design 1) is basically just an ArrayList.

Recall the representation invariant for the underlying data array in an ArrayList.

data is an array of items, never null.
The i-th item in the list is always stored in data[i].

1. How does maintaining this invariant affect the runtimes for add and remove?

2. Propose an invariant that could result in faster runtimes for add and remove.

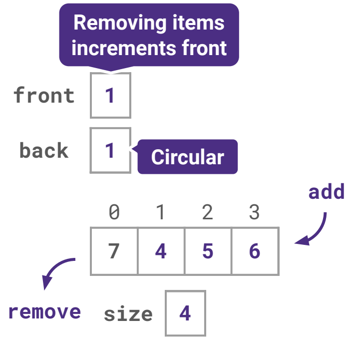
Q1: How does this invariant relate to the runtimes for add and remove?

Q2: Propose an invariant that could result in faster runtimes for add and remove.

A ArrayQueue: Design 2

The i-th item does not need to be data[i] so the front of the queue does not need to be the front of the array!

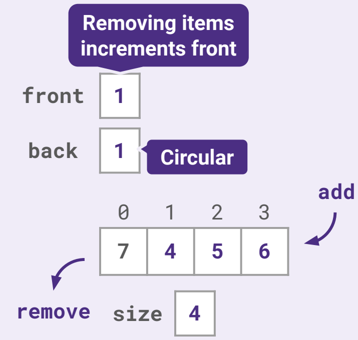
```
add(3)
add(4)
remove()
add(5)
add(6)
add(7)
```



Q ArrayQueue: Design 2

The i-th item does not need to be data[i] so the front of the queue does not need to be the front of the array!

Give an invariant that describes this behavior in your own words.



front represents the index of the front of the queue (except when the queue is empty) while back represents the index for the **next** item.

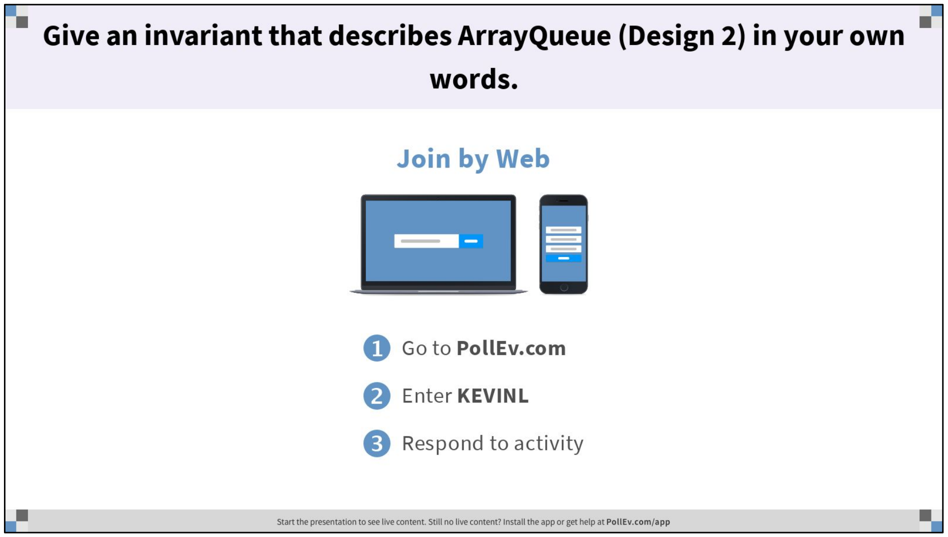
?: What's the runtime for ArrayQueue (Design 2) add and remove?

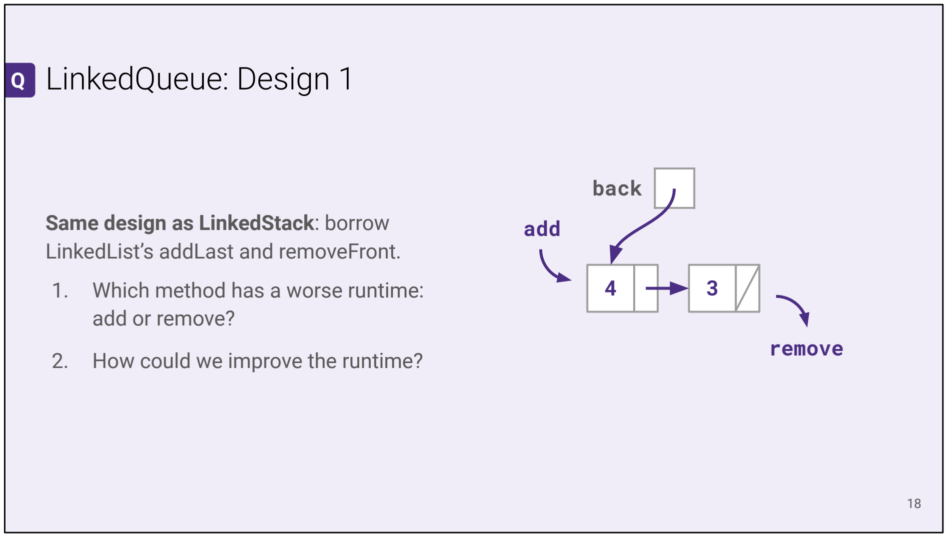
?: Is it necessary to maintain an integer index for remembering the back of the array?

?: We found a faster way to implement ArrayQueue. Is it possible to take these invariants and use them to implement a faster ArrayList?

front represents the index of the front of the queue (except when the queue is empty) while back represents the index for the **next** item.

Q1: Give an invariant that describes this behavior in your own words.

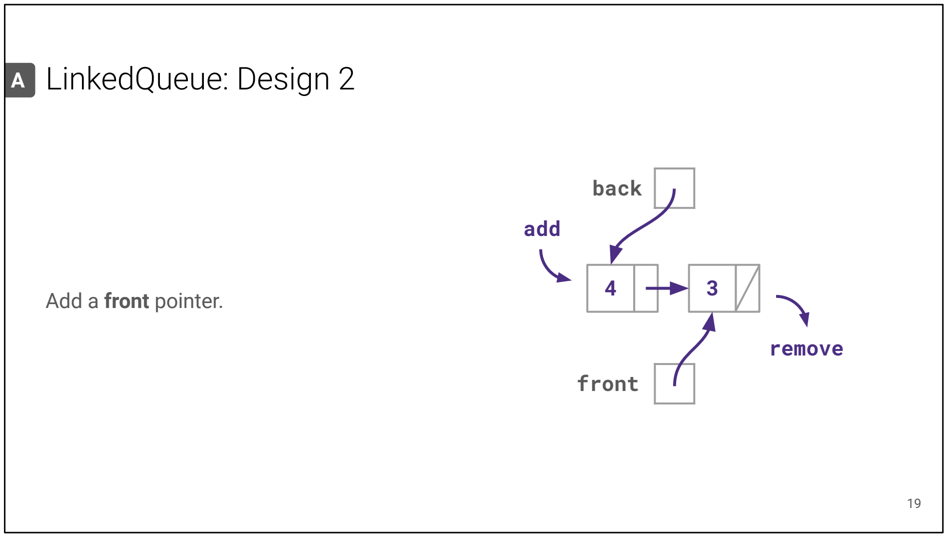




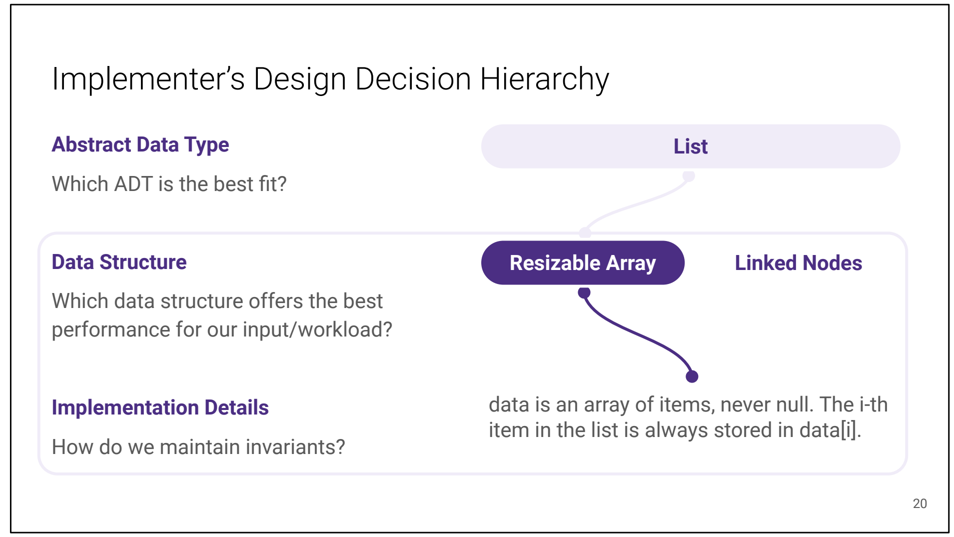
Q1: Which method has a worse runtime: add or remove?

Q2: How would you improve the runtime?

?: How does this change your visualization of the data structure?



?: What are other possible designs for LinkedQueue? What set of invariants can result in a **slower** LinkedQueue implementation?



Today, we studied the ADT implementer's view of the Design Decision Hierarchy. A recurring theme in computer science is that problem representations (**implementation details**) reflect problem solutions (**data structures**).

One neat observation: by simplifying the ADT interface, we gave the implementer more control over how they implemented their data structures. The more complex the ADT, the more restrictive the invariants, which means the implementer might not be able to make as many runtime optimizations.

?: We'll later look at the ADT client's perspective. How does the client determine which ADT is the best fit? To what extent does the client need to worry about ADT and data structure complexity?