ساختمان داده ها و الگوریتم ها(CE203)

جلسه دوازدهم:
لیست، پشته و صف

سجاد شیرعلی شهرضا

پاییز 1400

شنبه، 15 آبان 1400

Based on Washington University CSE 373 Slides from Autumn 2020 by Hunter Schafer

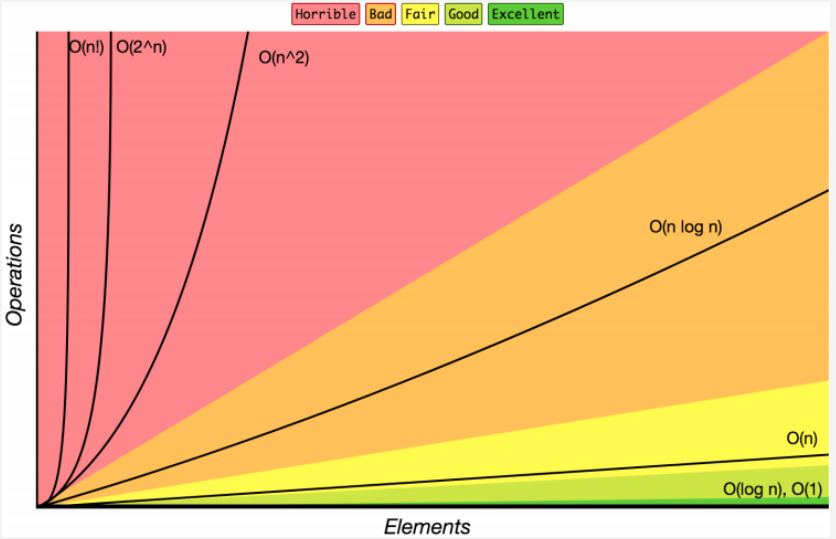
اطلاع رسانی

- بخش مرتبط کتاب برای این جلسه: 10
- مهلت ارسال تمرین دوم: شنبه هفته آینده، 22 آبان ساعت 8 صبح

Complexity Class

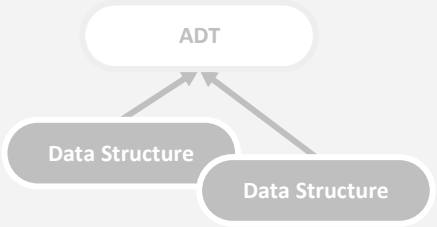
- **Complexity Class**: a category of algorithm efficiency based on the algorithm's relationship to the input size N

Complexity Class	Big-O	Runtime if you double N
constant	**O(1)**	**unchanged**
logarithmic	O($\log_2$ N)	increases slightly
linear	**O(N)**	**doubles**
log-linear	O(N $\log_2$ N)	slightly more than doubles
quadratic	**O(N^2)**	**quadruples**
…	…	…
exponential	O(2^N)	multiplies drastically

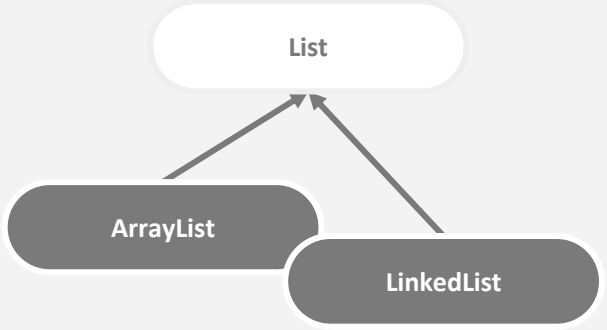


ADTs: Abstract Data Types

- An **abstract data type** is a data type that does not specify any one implementation.
  - Think of this as an agreement: about *what* is provided, but not *how*.



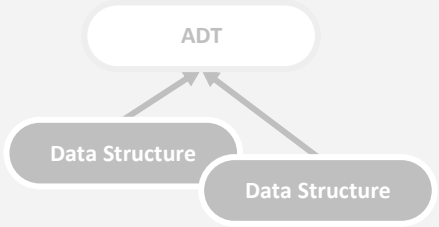
For Example:



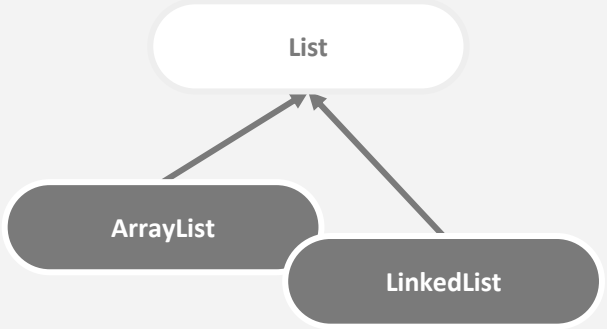
ADTs: Abstract Data Types

- An **abstract data type** is a data type that does not specify any one implementation.
  - Think of this as an agreement: about *what* is provided, but not *how*.

- **Data structures** implement ADTs.
  - **Resizable array** can implement List, Stack, Queue, Deque, PQ, etc.
  - **Linked nodes** can implement List, Stack, Queue, Deque, PQ, etc.



For Example:



لیست

مجموعه ای ترتیب دار از اشیاء

The List ADT

List: a collection storing an ordered sequence of elements.

- Each item is accessible by an index.
- A list has a variable size defined as the number of elements in the list
- Elements can be added to or removed from any position in the list

Relation to code/mental image of a list:

```
List<String> names = new ArrayList<>();     // []
names.size();                               // evaluates to 0
names.add(“Leona”);                         // [“Leona”]
names.add(“Ryan”);                          // [“Leona, Ryan”]
names.insert(“Paul”, 0);                    // [“Paul”, “Leona”, “Ryan”]
names.size();                               // evaluates to 3
```

List Implementations

LIST ADT

State

Set of ordered items
Count of items

Behavior

get(index) return item at index
set(item, index) replace item at index
add(item) add item to end of list
insert(item, index) add item at index
delete(index) delete item at index
size() count of items

ArrayList<E>

State

```
data[]
size
```

Behavior

```
get return data[index]
set data[index] = value
add data[size] = value, if out
of space grow data
insert shift values to make hole
at index, data[index] = value,
if out of space grow data
delete shift following values
forward
size return size
```

LinkedList<E>

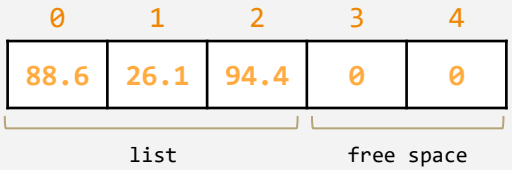
State

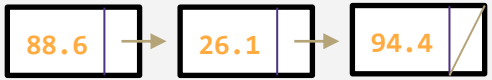
```
Node front;
size
```

Behavior

```
get loop until index, return node's
value
set loop until index, update node's
value
add create new node, update next of
last node
insert create new node, loop until
index, update next fields
delete loop until index, skip node
size return size
```

[88.6, 26.1, 94.4]





ArrayList

- How do Java / other programming languages implement ArrayList to achieve all the List behavior?
- On the inside:
  - stores the elements **inside an array** (which has a fixed capacity) that typically has more space than currently used (For example when there is only 1 element in the actual list, the array might have 10 spaces for data),

List View

```
["Paul", "Leona", "Ryan"]
```

ArrayList View

```
["Paul", "Leona", "Ryan", null, null, null]
```

ArrayList

- How do Java / other programming languages implement ArrayList to achieve all the List behavior?
- On the inside:
  - stores the elements **inside an array** (which has a fixed capacity) that typically has more space than currently used (For example when there is only 1 element in the actual list, the array might have 10 spaces for data),
  - stores all of these elements at the front of the array and **keeps track of how many there are** (the size) so that the implementation doesn't get confused enough to look at the empty space. This means that sometimes we will have to do a lot of work to shift the elements around.

List View

```
["Paul", "Leona", "Ryan"]
```

ArrayList View

```
["Paul", "Leona", "Ryan", null, null, null]
```

Implementing ArrayList

ArrayList<E>

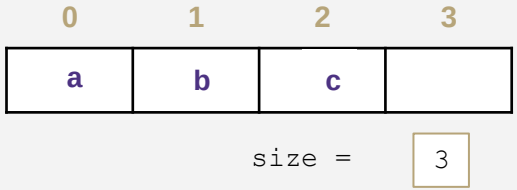
State

```
data[]
size
```

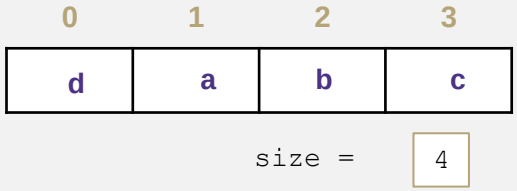
Behavior

```
get return data[index]
set data[index] = value
add data[size] = value, if out
of space grow data
insert shift values to make hole
at index, data[index] = value,
if out of space grow data
delete shift following values
forward
size return size
```

`insert(element, index)` with shifting



`delete(index)` with shifting



Implementing ArrayList

ArrayList<E>

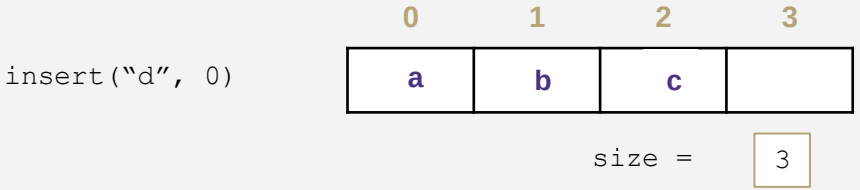
State

```
data[]
size
```

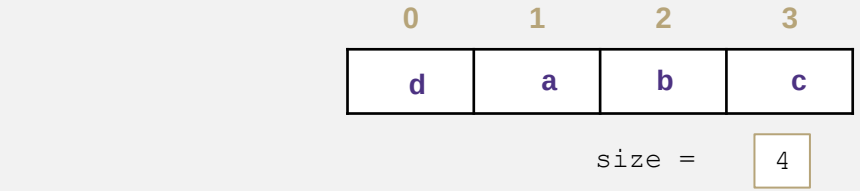
Behavior

```
get return data[index]
set data[index] = value
add data[size] = value, if out
of space grow data
insert shift values to make hole
at index, data[index] = value,
if out of space grow data
delete shift following values
forward
size return size
```

`insert(element, index)` with shifting



`delete(index)` with shifting



Implementing ArrayList

ArrayList<E>

State

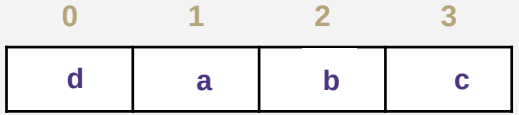
```
data[]
size
```

Behavior

```
get return data[index]
set data[index] = value
add data[size] = value, if out
of space grow data
insert shift values to make hole
at index, data[index] = value,
if out of space grow data
delete shift following values
forward
size return size
```

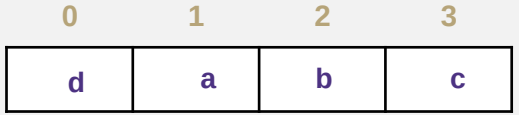
`insert(element, index)` with shifting

`insert("d", 0)`



`size =` 4

`delete(index)` with shifting



`size =` 4

Implementing ArrayList

ArrayList<E>

State

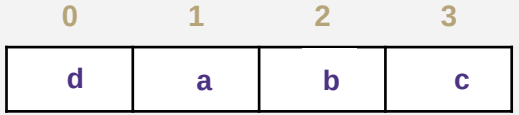
```
data[]
size
```

Behavior

```
get return data[index]
set data[index] = value
add data[size] = value, if out
of space grow data
insert shift values to make hole
at index, data[index] = value,
if out of space grow data
delete shift following values
forward
size return size
```

`insert(element, index)` with shifting

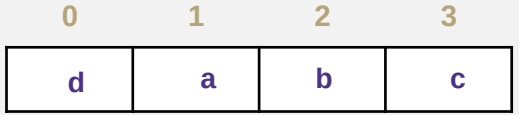
`insert("d", 0)`



`size =` 4

`delete(index)` with shifting

`delete(0)`



`size =` 4

Implementing ArrayList

ArrayList<E>

State

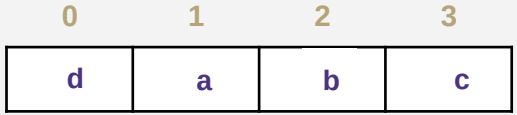
```
data[]
size
```

Behavior

```
get return data[index]
set data[index] = value
add data[size] = value, if out
of space grow data
insert shift values to make hole
at index, data[index] = value,
if out of space grow data
delete shift following values
forward
size return size
```

`insert(element, index)` with shifting

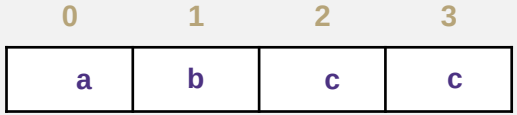
`insert("d", 0)`



`size =` 4

`delete(index)` with shifting

`delete(0)`



`size =` 3

Should we overwrite index 3 with null/0/-1?

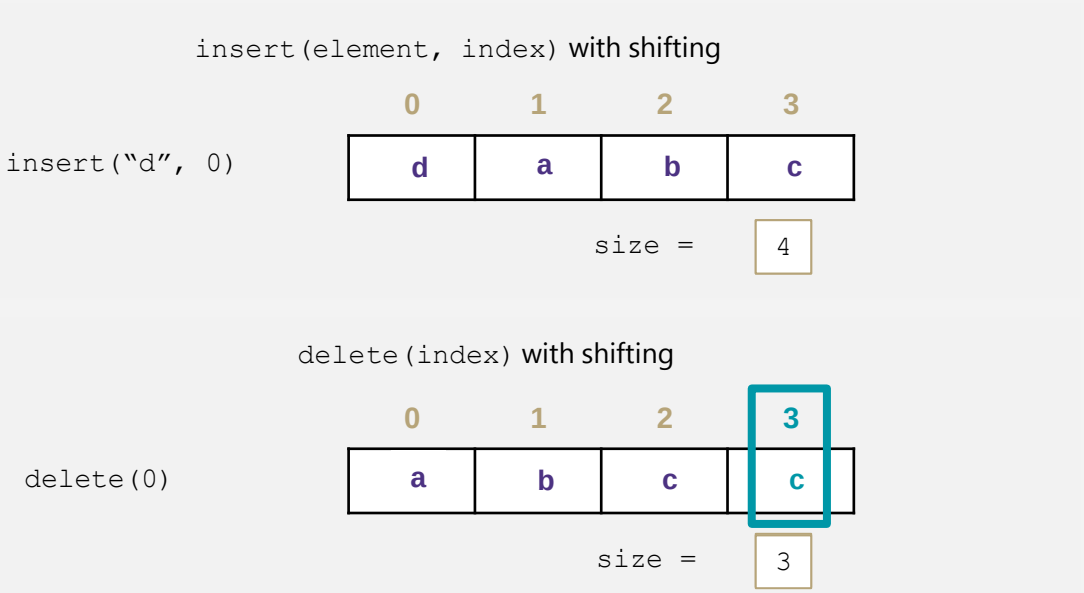
ArrayList<E>

State

- data[]
- size

Behavior

- get return data[index]
- set data[index] = value
- add data[size] = value, if out of space grow data
- insert shift values to make hole at index, data[index] = value, if out of space grow data
- delete shift following values forward
- size return size



Implementing ArrayList – Insert with Expansion

ArrayList<E>

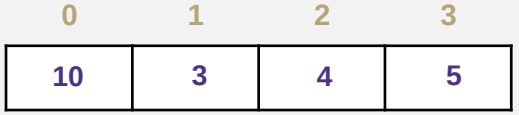
State

```
data[]
size
```

Behavior

```
get return data[index]
set data[index] = value
add data[size] = value, if out
of space grow data
insert shift values to make hole
at index, data[index] = value,
if out of space grow data
delete shift following values
forward
size return size
```

`append(element)` with growth



`numberOfItems =` 4

Implementing ArrayList – Insert with Expansion

ArrayList<E>

State

```
data[]
size
```

Behavior

```
get return data[index]
set data[index] = value
add data[size] = value, if out
of space grow data
insert shift values to make hole
at index, data[index] = value,
if out of space grow data
delete shift following values
forward
size return size
```

`append(element)` with growth

`append(2)`

0	1	2	3
10	3	4	5

`numberOfItems =` 4

Implementing ArrayList – Insert with Expansion

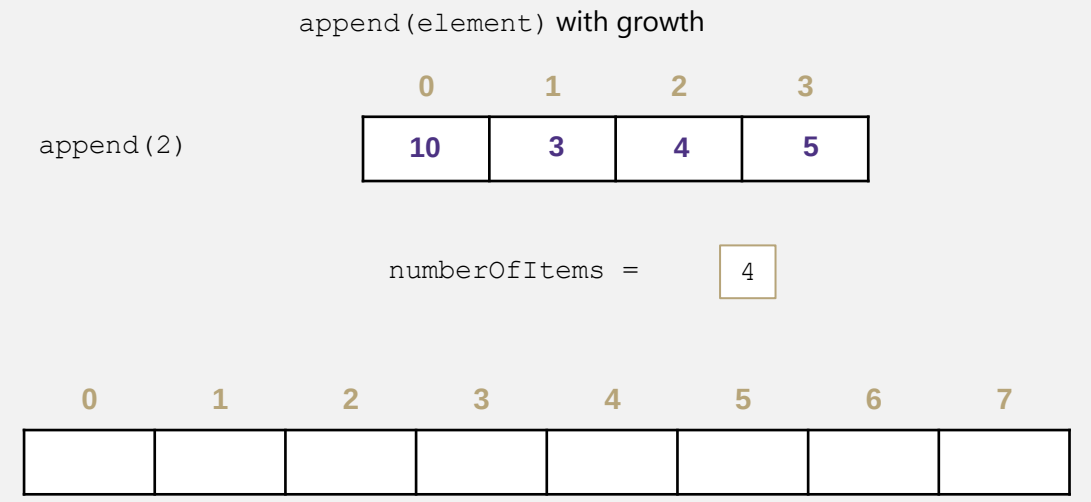
ArrayList<E>

State

```
data[]
size
```

Behavior

```
get return data[index]
set data[index] = value
add data[size] = value, if out
of space grow data
insert shift values to make hole
at index, data[index] = value,
if out of space grow data
delete shift following values
forward
size return size
```



Implementing ArrayList – Insert with Expansion

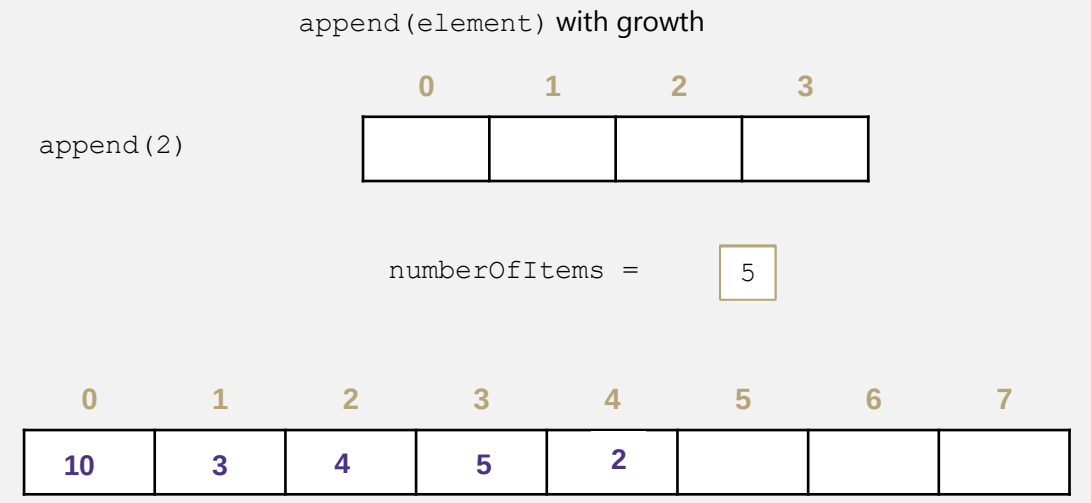
ArrayList<E>

State

- `data[]`
- `size`

Behavior

- `get` return data[index]
- `set` data[index] = value
- `add` data[size] = value, if out of space grow data
- `insert` shift values to make hole at index, data[index] = value, if out of space grow data
- `delete` shift following values forward
- `size` return size



Design Decisions

- For every ADT, many ways to implement
- Based on your situation you should consider:
    - Speed vs Memory Usage
    - Generic/Reusability vs Specific/Specialized
    - One Function vs Another
    - Robustness vs Performance
- **You job is selecting the best ADT implementing by making the right design tradeoffs!**
    - A common topic in interview questions

Design Decisions

- Akbar Joojeh is implementing a new system to manage orders
- When an order comes in, it’s placed at the end of the set of orders
- Food is prepared in approximately the same order it was requested, but sometimes orders are fulfilled out of order

- Let’s represent tickets using the List ADT.
  What implementation should we use? Why?

What implementation should we use? Why?

- ArrayList
  - Creating a new order is very fast (as long as we don’t have to resize)
  - Cooks can see any given order easily
- LinkedList
  - Creating an order is slower (have to iterate through whole list)
  - We’ll mostly be removing from the front of the list, which is fast because it requires no shifting

Comparing ADT Implementations: List

	ArrayList	LinkedList
`add (front)`	linear	constant
`remove (front)`	linear	constant
`add (back)`	(usually) constant	linear
`remove (back)`	constant	linear
`get`	constant	linear
`put`	linear	linear

Comparing ADT Implementations: List

	ArrayList	LinkedList
`add (front)`	linear	constant
`remove (front)`	linear	constant
`add (back)`	(usually) constant	linear
`remove (back)`	constant	linear
`get`	constant	linear
`put`	linear	linear

- Important to be able to come up with this, and understand why
- But only half the story: to be able to make a design decision, need the context to understand which of these we should prioritize

Design Decisions

- Both ArrayList and LinkedList have pros and cons, neither is strictly better than the other

- The Design Decision process:
  - **Evaluate** pros and cons
  - **Decide** on a design
  - **Defend** your design decision

- This is a major objective of the course!

سوال؟

پشته

مجموعه ای از اشیاء روی هم قرار گرفته

The Stack ADT

- **Stack**: an ADT representing an ordered sequence of elements whose elements can only be added & removed from one end.
  - Last-In, First-Out (LIFO)
  - Elements stored in order of insertion
    - We don't think of them as having indices
  - Clients can only add/remove/examine the "top"

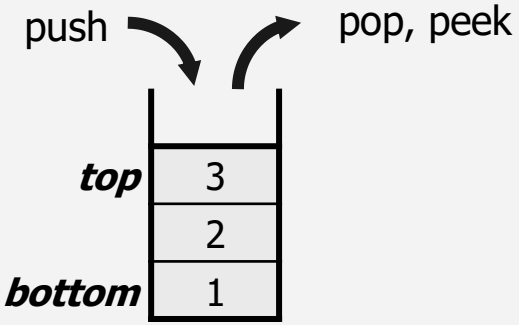
STACK ADT

State

Collection of ordered items
Count of items

Behavior

`push(index)` add item to top
`pop()` return & remove item at top
`peek()` return item at top
`size()` count of items
`isEmpty()` is count 0?



Implementing a Stack with Linked Nodes

STACK ADT

State

Collection of ordered items
Count of items

Behavior

`push(index)` add item to top
`pop()` return & remove item at top
`peek()` return item at top
`size()` count of items
`isEmpty()` is count 0?

LinkedStack<E>

State

`Node top`
`size`

Behavior

`push` add new node at top
`pop` return & remove node at top
`peek` return node at top
`size` return size
`isEmpty` return size == 0

Implementing a Stack with Linked Nodes

STACK ADT

State

Collection of ordered items
Count of items

Behavior

`push(index)` add item to top
`pop()` return & remove item at top
`peek()` return item at top
`size()` count of items
`isEmpty()` is count 0?

LinkedStack<E>

State

`Node top`
`size`

Behavior

`push` add new node at top
`pop` return & remove node at top
`peek` return node at top
`size` return size
`isEmpty` return size == 0

Implementing a Stack with Linked Nodes

STACK ADT

State

Collection of ordered items
Count of items

Behavior

`push(index)` add item to top
`pop()` return & remove item at top
`peek()` return item at top
`size()` count of items
`isEmpty()` is count 0?

LinkedStack<E>

State

`Node top`
`size`

Behavior

`push` add new node at top
`pop` return & remove node at top
`peek` return node at top
`size` return size
`isEmpty` return size == 0

`push(3)`

`size =` 0

Implementing a Stack with Linked Nodes

STACK ADT

State

Collection of ordered items
Count of items

Behavior

`push(index)` add item to top
`pop()` return & remove item at top
`peek()` return item at top
`size()` count of items
`isEmpty()` is count 0?

LinkedStack<E>

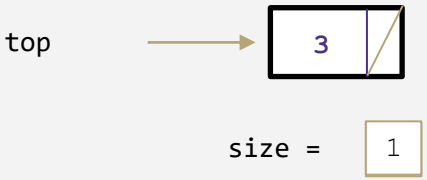
State

`Node top`
`size`

Behavior

`push` add new node at top
`pop` return & remove node at top
`peek` return node at top
`size` return size
`isEmpty` return size == 0

`push(3)`



Implementing a Stack with Linked Nodes

STACK ADT

State

Collection of ordered items
Count of items

Behavior

`push(index)` add item to top
`pop()` return & remove item at top
`peek()` return item at top
`size()` count of items
`isEmpty()` is count 0?

LinkedStack<E>

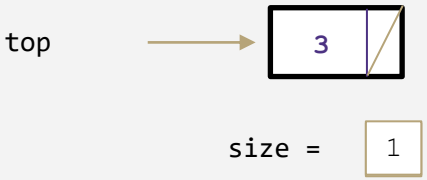
State

`Node top`
`size`

Behavior

`push` add new node at top
`pop` return & remove node at top
`peek` return node at top
`size` return size
`isEmpty` return size == 0

```
push(3)
push(4)
```



Implementing a Stack with Linked Nodes

STACK ADT

State

Collection of ordered items
Count of items

Behavior

`push(index)` add item to top
`pop()` return & remove item at top
`peek()` return item at top
`size()` count of items
`isEmpty()` is count 0?

LinkedStack<E>

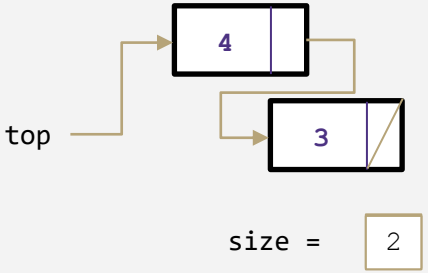
State

`Node top`
`size`

Behavior

`push` add new node at top
`pop` return & remove node at top
`peek` return node at top
`size` return size
`isEmpty` return size == 0

```
push(3)
push(4)
```



Implementing a Stack with Linked Nodes

STACK ADT

State

Collection of ordered items
Count of items

Behavior

`push(index)` add item to top
`pop()` return & remove item at top
`peek()` return item at top
`size()` count of items
`isEmpty()` is count 0?

LinkedStack<E>

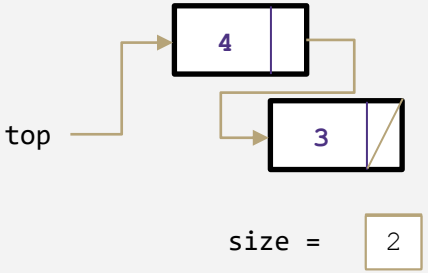
State

`Node top`
`size`

Behavior

`push` add new node at top
`pop` return & remove node at top
`peek` return node at top
`size` return size
`isEmpty` return size == 0

```
push(3)
push(4)
pop()
```



Implementing a Stack with Linked Nodes

STACK ADT

State

Collection of ordered items
Count of items

Behavior

`push(index)` add item to top
`pop()` return & remove item at top
`peek()` return item at top
`size()` count of items
`isEmpty()` is count 0?

LinkedStack<E>

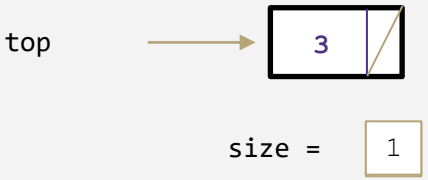
State

`Node top`
`size`

Behavior

`push` add new node at top
`pop` return & remove node at top
`peek` return node at top
`size` return size
`isEmpty` return size == 0

```
push(3)
push(4)
pop()
```



Implementing a Stack with Linked Nodes

STACK ADT

State

Collection of ordered items
Count of items

Behavior

`push(index)` add item to top
`pop()` return & remove item at top
`peek()` return item at top
`size()` count of items
`isEmpty()` is count 0?

LinkedStack<E>

State

`Node top`
`size`

Behavior

`push` add new node at top
`pop` return & remove node at top
`peek` return node at top
`size` return size
`isEmpty` return size == 0

Big-Oh Analysis

`pop()`

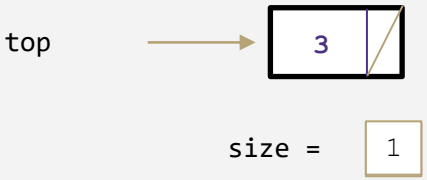
`peek()`

`size()`

`isEmpty()`

`push()`

```
push(3)
push(4)
pop()
```



Implementing a Stack with Linked Nodes

STACK ADT

State

Collection of ordered items
Count of items

Behavior

`push(index)` add item to top
`pop()` return & remove item at top
`peek()` return item at top
`size()` count of items
`isEmpty()` is count 0?

LinkedStack<E>

State

`Node top`
`size`

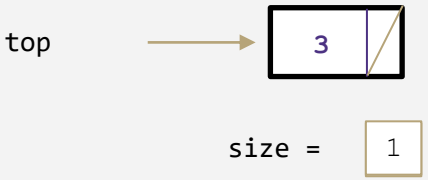
Behavior

`push` add new node at top
`pop` return & remove node at top
`peek` return node at top
`size` return size
`isEmpty` return size == 0

Big-Oh Analysis

`pop()`	O(1) Constant
`peek()`	
`size()`	
`isEmpty()`	
`push()`	

```
push(3)
push(4)
pop()
```



Implementing a Stack with Linked Nodes

STACK ADT

State

Collection of ordered items
Count of items

Behavior

`push(index)` add item to top
`pop()` return & remove item at top
`peek()` return item at top
`size()` count of items
`isEmpty()` is count 0?

LinkedStack<E>

State

`Node top`
`size`

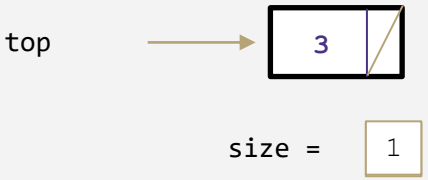
Behavior

`push` add new node at top
`pop` return & remove node at top
`peek` return node at top
`size` return size
`isEmpty` return size == 0

Big-Oh Analysis

`pop()`	O(1) Constant
`peek()`	O(1) Constant
`size()`	
`isEmpty()`	
`push()`	

```
push(3)
push(4)
pop()
```



Implementing a Stack with Linked Nodes

STACK ADT

State

Collection of ordered items
Count of items

Behavior

`push(index)` add item to top
`pop()` return & remove item at top
`peek()` return item at top
`size()` count of items
`isEmpty()` is count 0?

LinkedStack<E>

State

`Node top`
`size`

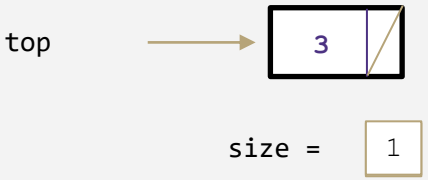
Behavior

`push` add new node at top
`pop` return & remove node at top
`peek` return node at top
`size` return size
`isEmpty` return size == 0

Big-Oh Analysis

`pop()`	O(1) Constant
`peek()`	O(1) Constant
`size()`	O(1) Constant
`isEmpty()`	
`push()`	

```
push(3)
push(4)
pop()
```



Implementing a Stack with Linked Nodes

STACK ADT

State

Collection of ordered items
Count of items

Behavior

`push(index)` add item to top
`pop()` return & remove item at top
`peek()` return item at top
`size()` count of items
`isEmpty()` is count 0?

LinkedStack<E>

State

`Node top`
`size`

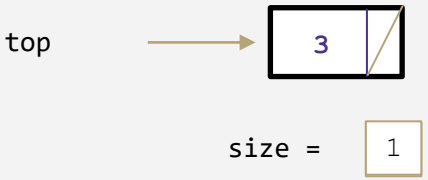
Behavior

`push` add new node at top
`pop` return & remove node at top
`peek` return node at top
`size` return size
`isEmpty` return size == 0

Big-Oh Analysis

`pop()`	O(1) Constant
`peek()`	O(1) Constant
`size()`	O(1) Constant
`isEmpty()`	O(1) Constant
`push()`	

```
push(3)
push(4)
pop()
```



Implementing a Stack with Linked Nodes

STACK ADT

State

Collection of ordered items
Count of items

Behavior

`push(index)` add item to top
`pop()` return & remove item at top
`peek()` return item at top
`size()` count of items
`isEmpty()` is count 0?

LinkedStack<E>

State

`Node top`
`size`

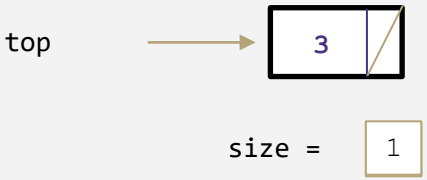
Behavior

`push` add new node at top
`pop` return & remove node at top
`peek` return node at top
`size` return size
`isEmpty` return size == 0

Big-Oh Analysis

`pop()`	O(1) Constant
`peek()`	O(1) Constant
`size()`	O(1) Constant
`isEmpty()`	O(1) Constant
`push()`	

```
push(3)
push(4)
pop()
```



What do you think the worst possible runtime of `push()` could be?

Implementing a Stack with Linked Nodes

STACK ADT

State

Collection of ordered items
Count of items

Behavior

`push(index)` add item to top
`pop()` return & remove item at top
`peek()` return item at top
`size()` count of items
`isEmpty()` is count 0?

LinkedStack<E>

State

`Node top`
`size`

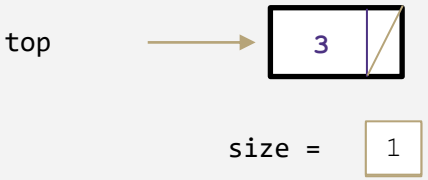
Behavior

`push` add new node at top
`pop` return & remove node at top
`peek` return node at top
`size` return size
`isEmpty` return size == 0

Big-Oh Analysis

`pop()`	O(1) Constant
`peek()`	O(1) Constant
`size()`	O(1) Constant
`isEmpty()`	O(1) Constant
`push()`	O(1) otherwise

```
push(3)
push(4)
pop()
```



سوال؟

Implementing a Stack with an Array

STACK ADT

State

Collection of ordered items
Count of items

Behavior

`push(index)` add item to top
`pop()` return & remove item at top
`peek()` return item at top
`size()` count of items
`isEmpty()` is count 0?

ArrayStack<E>

State

`data[]`
`size`

Behavior

`push` data[size] = value, if out of room grow data
`pop` return data[size - 1], size -= 1
`peek` return data[size - 1]
`size` return size
`isEmpty` return size == 0

Implementing a Stack with an Array

STACK ADT

State

Collection of ordered items
Count of items

Behavior

`push(index)` add item to top
`pop()` return & remove item at top
`peek()` return item at top
`size()` count of items
`isEmpty()` is count 0?

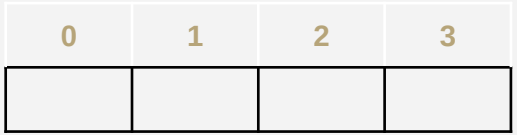
ArrayStack<E>

State

`data[]`
`size`

Behavior

`push` data[size] = value, if out of room grow data
`pop` return data[size - 1], size -= 1
`peek` return data[size - 1]
`size` return size
`isEmpty` return size == 0



size = 0

Implementing a Stack with an Array

STACK ADT

State

Collection of ordered items
Count of items

Behavior

`push(index)` add item to top
`pop()` return & remove item at top
`peek()` return item at top
`size()` count of items
`isEmpty()` is count 0?

ArrayStack<E>

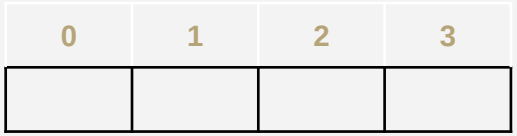
State

`data[]`
`size`

Behavior

`push` data[size] = value, if out of room grow data
`pop` return data[size - 1], size -= 1
`peek` return data[size - 1]
`size` return size
`isEmpty` return size == 0

`push(3)`



`size =` 0

Implementing a Stack with an Array

STACK ADT

State

Collection of ordered items
Count of items

Behavior

`push(index)` add item to top
`pop()` return & remove item at top
`peek()` return item at top
`size()` count of items
`isEmpty()` is count 0?

ArrayStack<E>

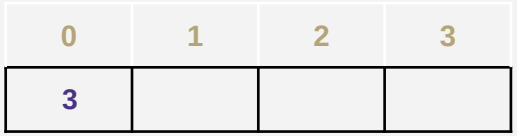
State

`data[]`
`size`

Behavior

`push` data[size] = value, if out of room grow data
`pop` return data[size - 1], size -= 1
`peek` return data[size - 1]
`size` return size
`isEmpty` return size == 0

`push(3)`



size = 1

Implementing a Stack with an Array

STACK ADT

State

Collection of ordered items
Count of items

Behavior

`push(index)` add item to top
`pop()` return & remove item at top
`peek()` return item at top
`size()` count of items
`isEmpty()` is count 0?

ArrayStack<E>

State

`data[]`
`size`

Behavior

`push` data[size] = value, if out of room grow data
`pop` return data[size - 1], size -= 1
`peek` return data[size - 1]
`size` return size
`isEmpty` return size == 0

```
push(3)
push(4)
```

0	1	2	3
3			

size = 1

Implementing a Stack with an Array

STACK ADT

State

Collection of ordered items
Count of items

Behavior

`push(index)` add item to top
`pop()` return & remove item at top
`peek()` return item at top
`size()` count of items
`isEmpty()` is count 0?

ArrayStack<E>

State

`data[]`
`size`

Behavior

`push` data[size] = value, if out of room grow data
`pop` return data[size - 1], size -= 1
`peek` return data[size - 1]
`size` return size
`isEmpty` return size == 0

```
push(3)
push(4)
```

0	1	2	3
3	4		

size = 2

Implementing a Stack with an Array

STACK ADT

State

Collection of ordered items
Count of items

Behavior

`push(index)` add item to top
`pop()` return & remove item at top
`peek()` return item at top
`size()` count of items
`isEmpty()` is count 0?

ArrayStack<E>

State

`data[]`
`size`

Behavior

`push` data[size] = value, if out of room grow data
`pop` return data[size - 1], size -= 1
`peek` return data[size - 1]
`size` return size
`isEmpty` return size == 0

```
push(3)
push(4)
pop()
```

0	1	2	3
3	4		

size = 2

Implementing a Stack with an Array

STACK ADT

State

Collection of ordered items
Count of items

Behavior

`push(index)` add item to top
`pop()` return & remove item at top
`peek()` return item at top
`size()` count of items
`isEmpty()` is count 0?

ArrayStack<E>

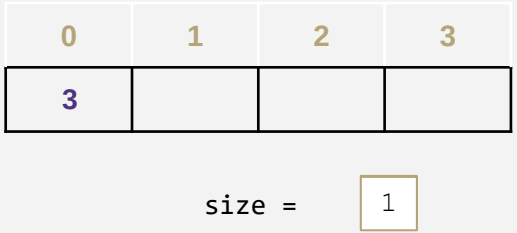
State

`data[]`
`size`

Behavior

`push` data[size] = value, if out of room grow data
`pop` return data[size - 1], size -= 1
`peek` return data[size - 1]
`size` return size
`isEmpty` return size == 0

```
push(3)
push(4)
pop()
```



Implementing a Stack with an Array

STACK ADT

State

Collection of ordered items
Count of items

Behavior

`push(index)` add item to top
`pop()` return & remove item at top
`peek()` return item at top
`size()` count of items
`isEmpty()` is count 0?

ArrayStack<E>

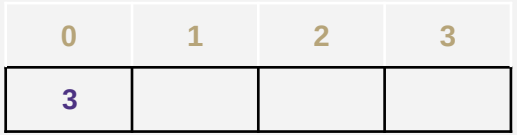
State

`data[]`
`size`

Behavior

`push` data[size] = value, if out of room grow data
`pop` return data[size - 1], size -= 1
`peek` return data[size - 1]
`size` return size
`isEmpty` return size == 0

```
push(3)
push(4)
pop()
push(5)
```



size = 1

Implementing a Stack with an Array

STACK ADT

State

Collection of ordered items
Count of items

Behavior

`push(index)` add item to top
`pop()` return & remove item at top
`peek()` return item at top
`size()` count of items
`isEmpty()` is count 0?

ArrayStack<E>

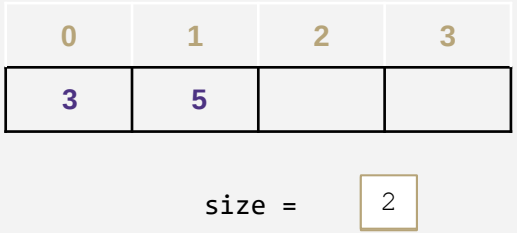
State

`data[]`
`size`

Behavior

`push` data[size] = value, if out of room grow data
`pop` return data[size - 1], size -= 1
`peek` return data[size - 1]
`size` return size
`isEmpty` return size == 0

```
push(3)
push(4)
pop()
push(5)
```



Implementing a Stack with an Array

STACK ADT

State

Collection of ordered items
Count of items

Behavior

push(index) add item to top
pop() return & remove item at top
peek() return item at top
size() count of items
isEmpty() is count 0?

ArrayStack<E>

State

data[]
size

Behavior

push data[size] = value, if out of room grow data
pop return data[size - 1], size -= 1
peek return data[size - 1]
size return size
isEmpty return size == 0

Big-Oh Analysis

pop()

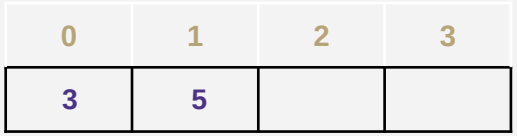
peek()

size()

isEmpty()

push()

```
push(3)
push(4)
pop()
push(5)
```



size = 2

Implementing a Stack with an Array

STACK ADT

State

Collection of ordered items
Count of items

Behavior

push(index) add item to top
pop() return & remove item at top
peek() return item at top
size() count of items
isEmpty() is count 0?

ArrayStack<E>

State

data[]
size

Behavior

push data[size] = value, if out of room grow data
pop return data[size - 1], size -= 1
peek return data[size - 1]
size return size
isEmpty return size == 0

Big-Oh Analysis

pop()	O(1) Constant
peek()	
size()	
isEmpty()	
push()	

```
push(3)
push(4)
pop()
push(5)
```

0	1	2	3
3	5		

size = 2

Implementing a Stack with an Array

STACK ADT

State

Collection of ordered items
Count of items

Behavior

`push(index)` add item to top
`pop()` return & remove item at top
`peek()` return item at top
`size()` count of items
`isEmpty()` is count 0?

ArrayStack<E>

State

`data[]`
`size`

Behavior

`push` data[size] = value, if out of room grow data
`pop` return data[size - 1], size -= 1
`peek` return data[size - 1]
`size` return size
`isEmpty` return size == 0

Big-Oh Analysis

pop()	O(1) Constant
peek()	O(1) Constant
size()	
isEmpty()	
push()	

```
push(3)
push(4)
pop()
push(5)
```

0	1	2	3
3	5		

size = 2

Implementing a Stack with an Array

STACK ADT

State

Collection of ordered items
Count of items

Behavior

`push(index)` add item to top
`pop()` return & remove item at top
`peek()` return item at top
`size()` count of items
`isEmpty()` is count 0?

ArrayStack<E>

State

`data[]`
`size`

Behavior

`push` data[size] = value, if out of room grow data
`pop` return data[size - 1], size -= 1
`peek` return data[size - 1]
`size` return size
`isEmpty` return size == 0

Big-Oh Analysis

`pop()`	O(1) Constant
`peek()`	O(1) Constant
`size()`	O(1) Constant
`isEmpty()`	
`push()`	

```
push(3)
push(4)
pop()
push(5)
```

0	1	2	3
3	5		

size = 2

Implementing a Stack with an Array

STACK ADT

State

Collection of ordered items
Count of items

Behavior

`push(index)` add item to top
`pop()` return & remove item at top
`peek()` return item at top
`size()` count of items
`isEmpty()` is count 0?

ArrayStack<E>

State

`data[]`
`size`

Behavior

`push` data[size] = value, if out of room grow data
`pop` return data[size - 1], size -= 1
`peek` return data[size - 1]
`size` return size
`isEmpty` return size == 0

Big-Oh Analysis

`pop()`	O(1) Constant
`peek()`	O(1) Constant
`size()`	O(1) Constant
`isEmpty()`	O(1) Constant
`push()`	

```
push(3)
push(4)
pop()
push(5)
```

0	1	2	3
3	5		

size = 2

Implementing a Stack with an Array

STACK ADT

State

Collection of ordered items
Count of items

Behavior

`push(index)` add item to top
`pop()` return & remove item at top
`peek()` return item at top
`size()` count of items
`isEmpty()` is count 0?

ArrayStack<E>

State

`data[]`
`size`

Behavior

`push` data[size] = value, if out of room grow data
`pop` return data[size - 1], size -= 1
`peek` return data[size - 1]
`size` return size
`isEmpty` return size == 0

Big-Oh Analysis

`pop()`	O(1) Constant
`peek()`	O(1) Constant
`size()`	O(1) Constant
`isEmpty()`	O(1) Constant
`push()`	

```
push(3)
push(4)
pop()
push(5)
```

0	1	2	3
3	5		

size = 2

What do you think the worst possible runtime of `push()` could be?

Implementing a Stack with an Array

STACK ADT

State

Collection of ordered items
Count of items

Behavior

`push(index)` add item to top
`pop()` return & remove item at top
`peek()` return item at top
`size()` count of items
`isEmpty()` is count 0?

ArrayStack<E>

State

`data[]`
`size`

Behavior

`push` data[size] = value, if out of room grow data
`pop` return data[size - 1], size -= 1
`peek` return data[size - 1]
`size` return size
`isEmpty` return size == 0

```
push(3)
push(4)
pop()
push(5)
```

0	1	2	3
3	5		

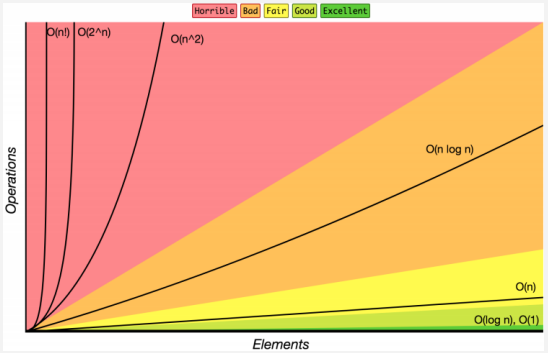
size = 2

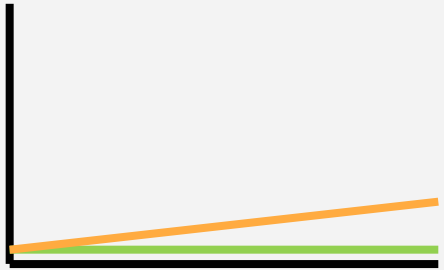
Big-Oh Analysis

`pop()`	O(1) Constant
`peek()`	O(1) Constant
`size()`	O(1) Constant
`isEmpty()`	O(1) Constant
`push()`	O(n) linear if you have to resize, O(1) otherwise

Why Not Decide on One?

- Big-Oh analysis of `push()`: **O(n) linear** if you have to resize,
  O(1) constant otherwise

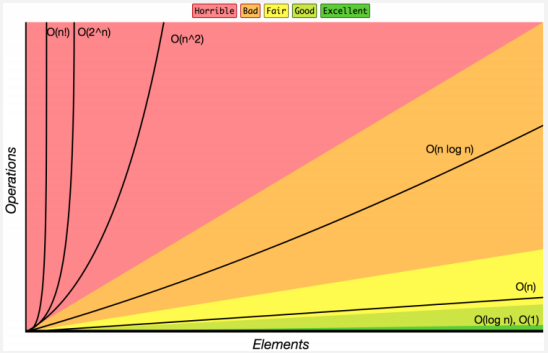


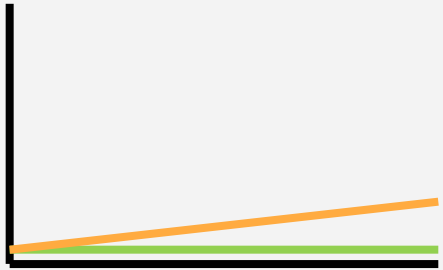


If you have to resize
Otherwise

Why Not Decide on One?

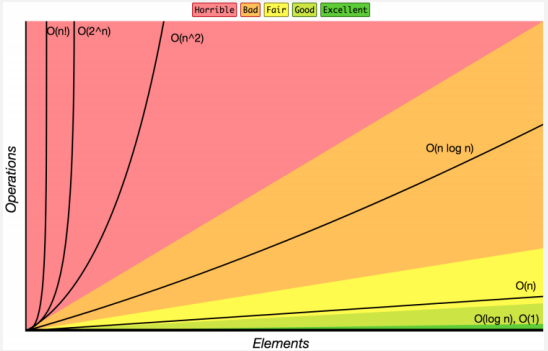
- Big-Oh analysis of `push()`: **O(n) linear** if you have to resize, **O(1) constant** otherwise
- Two insights to keep in mind:
  - Behavior is *completely* different in these two cases. Almost better not to try and analyze them both together.

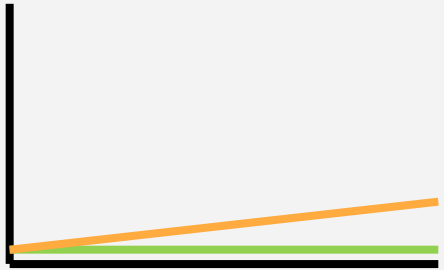




Why Not Decide on One?

- Big-Oh analysis of `push()`: **O(n) linear** if you have to resize, **O(1) constant** otherwise
- Two insights to keep in mind:
  - Behavior is *completely* different in these two cases. Almost better not to try and analyze them both together.
  - Big-Oh is a *tool* to describe runtime. Having to decide just one or the other would make it a less useful tool – not a complete description.





If you have to resize
Otherwise

سوال؟

صف

مجموعه ای از اشیاء پشت سر هم قرار گرفته

The Queue ADT

- **Queue**: an ADT representing an ordered sequence of elements whose elements can only be added from one end and removed from the other.
  - First-In, First-Out (FIFO)
  - Elements stored in order of insertion
    - We don't think of them as having indices
  - Clients can only add to the "end", and can only examine/remove at the "front"

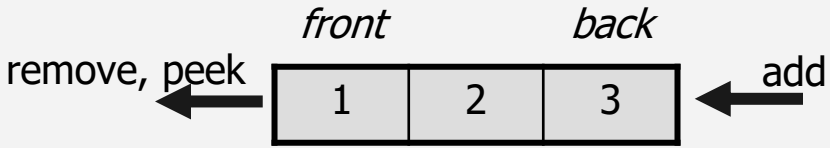
QUEUE ADT

State

Collection of ordered items
Count of items

Behavior

`add(item)` add item to back
`remove()` remove and return item at front
`peek()` return item at front
`size()` count of items
`isEmpty()` count is 0?



Implementing a Queue with Linked Nodes

QUEUE ADT

State

Collection of ordered items
Count of items

Behavior

`add(item)` add item to back
`remove()` remove and return item at front
`peek()` return item at front
`size()` count of items
`isEmpty()` count is 0?

LinkedQueue<E>

State

`Node front`
`Node back`
`size`

Behavior

`add` – add node to back
`remove` – return and remove node at front
`peek` – return node at front
`size` – return size
`isEmpty` – return size == 0

Implementing a Queue with Linked Nodes

QUEUE ADT

State

Collection of ordered items
Count of items

Behavior

`add(item)` add item to back
`remove()` remove and return item at front
`peek()` return item at front
`size()` count of items
`isEmpty()` count is 0?

LinkedQueue<E>

State

`Node front`
`Node back`
`size`

Behavior

`add` – add node to back
`remove` – return and remove node at front
`peek` – return node at front
`size` – return size
`isEmpty` – return size == 0

`size =` 0

`front` →

`back` →

Implementing a Queue with Linked Nodes

QUEUE ADT

State

Collection of ordered items
Count of items

Behavior

add(item) add item to back
remove() remove and return item at front
peek() return item at front
size() count of items
isEmpty() count is 0?

LinkedQueue<E>

State

Node front
Node back
size

Behavior

add – add node to back
remove – return and remove node at front
peek – return node at front
size – return size
isEmpty – return size == 0

size = 0

add(5)

front →

back →

Implementing a Queue with Linked Nodes

QUEUE ADT

State

Collection of ordered items
Count of items

Behavior

`add(item)` add item to back
`remove()` remove and return item at front
`peek()` return item at front
`size()` count of items
`isEmpty()` count is 0?

LinkedQueue<E>

State

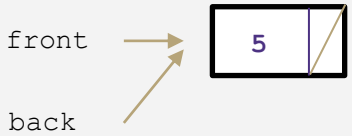
```
Node front
Node back
size
```

Behavior

`add` – add node to back
`remove` – return and remove node at front
`peek` – return node at front
`size` – return size
`isEmpty` – return size == 0

`size =` 1

`add(5)`



Implementing a Queue with Linked Nodes

QUEUE ADT

State

Collection of ordered items
Count of items

Behavior

`add(item)` add item to back
`remove()` remove and return item at front
`peek()` return item at front
`size()` count of items
`isEmpty()` count is 0?

LinkedQueue<E>

State

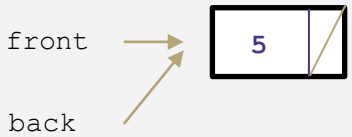
```
Node front
Node back
size
```

Behavior

`add` – add node to back
`remove` – return and remove node at front
`peek` – return node at front
`size` – return size
`isEmpty` – return size == 0

`size =` 1

```
add(5)
add(8)
```



Implementing a Queue with Linked Nodes

QUEUE ADT

State

Collection of ordered items
Count of items

Behavior

`add(item)` add item to back
`remove()` remove and return item at front
`peek()` return item at front
`size()` count of items
`isEmpty()` count is 0?

LinkedQueue<E>

State

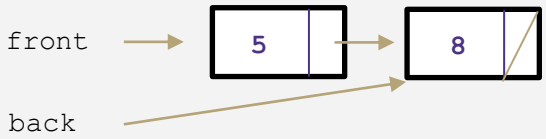
```
Node front
Node back
size
```

Behavior

`add` – add node to back
`remove` – return and remove node at front
`peek` – return node at front
`size` – return size
`isEmpty` – return size == 0

`size =` 2

```
add(5)
add(8)
```



Implementing a Queue with Linked Nodes

QUEUE ADT

State

Collection of ordered items
Count of items

Behavior

`add(item)` add item to back
`remove()` remove and return item at front
`peek()` return item at front
`size()` count of items
`isEmpty()` count is 0?

LinkedQueue<E>

State

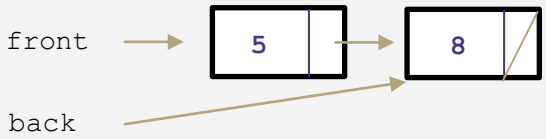
```
Node front
Node back
size
```

Behavior

`add` – add node to back
`remove` – return and remove node at front
`peek` – return node at front
`size` – return size
`isEmpty` – return size == 0

`size =` 2

```
add(5)
add(8)
remove()
```



Implementing a Queue with Linked Nodes

QUEUE ADT

State

Collection of ordered items
Count of items

Behavior

`add(item)` add item to back
`remove()` remove and return item at front
`peek()` return item at front
`size()` count of items
`isEmpty()` count is 0?

LinkedQueue<E>

State

```
Node front
Node back
size
```

Behavior

`add` – add node to back
`remove` – return and remove node at front
`peek` – return node at front
`size` – return size
`isEmpty` – return size == 0

`size =` 1

```
add(5)
add(8)
remove()
```

Implementing a Queue with Linked Nodes

QUEUE ADT

State

Collection of ordered items
Count of items

Behavior

`add(item)` add item to back
`remove()` remove and return item at front
`peek()` return item at front
`size()` count of items
`isEmpty()` count is 0?

LinkedQueue<E>

State

```
Node front
Node back
size
```

Behavior

`add` – add node to back
`remove` – return and remove node at front
`peek` – return node at front
`size` – return size
`isEmpty` – return size == 0

Big-Oh Analysis

`remove()`

`peek()`

`size()`

`isEmpty()`

`add()`

`size =` 1

```
add(5)
add(8)
remove()
```

Implementing a Queue with Linked Nodes

QUEUE ADT

State

Collection of ordered items
Count of items

Behavior

`add(item)` add item to back
`remove()` remove and return item at front
`peek()` return item at front
`size()` count of items
`isEmpty()` count is 0?

LinkedQueue<E>

State

```
Node front
Node back
size
```

Behavior

`add` – add node to back
`remove` – return and remove node at front
`peek` – return node at front
`size` – return size
`isEmpty` – return size == 0

Big-Oh Analysis

`remove()`	O(1) Constant
`peek()`	
`size()`	
`isEmpty()`	
`add()`	

`size =` 1

```
add(5)
add(8)
remove()
```

Implementing a Queue with Linked Nodes

QUEUE ADT

State

Collection of ordered items
Count of items

Behavior

`add(item)` add item to back
`remove()` remove and return item at front
`peek()` return item at front
`size()` count of items
`isEmpty()` count is 0?

LinkedQueue<E>

State

```
Node front
Node back
size
```

Behavior

`add` – add node to back
`remove` – return and remove node at front
`peek` – return node at front
`size` – return size
`isEmpty` – return size == 0

Big-Oh Analysis

`remove()`	O(1) Constant
`peek()`	O(1) Constant
`size()`	
`isEmpty()`	
`add()`	

`size =` 1

```
add(5)
add(8)
remove()
```

Implementing a Queue with Linked Nodes

QUEUE ADT

State

Collection of ordered items
Count of items

Behavior

`add(item)` add item to back
`remove()` remove and return item at front
`peek()` return item at front
`size()` count of items
`isEmpty()` count is 0?

LinkedQueue<E>

State

```
Node front
Node back
size
```

Behavior

`add` – add node to back
`remove` – return and remove node at front
`peek` – return node at front
`size` – return size
`isEmpty` – return size == 0

`size =` 1

Big-Oh Analysis

`remove()`	O(1) Constant
`peek()`	O(1) Constant
`size()`	O(1) Constant
`isEmpty()`	
`add()`	

```
add(5)
add(8)
remove()
```

Implementing a Queue with Linked Nodes

QUEUE ADT

State

Collection of ordered items
Count of items

Behavior

`add(item)` add item to back
`remove()` remove and return item at front
`peek()` return item at front
`size()` count of items
`isEmpty()` count is 0?

LinkedQueue<E>

State

```
Node front
Node back
size
```

Behavior

`add` – add node to back
`remove` – return and remove node at front
`peek` – return node at front
`size` – return size
`isEmpty` – return size == 0

Big-Oh Analysis

`remove()`	O(1) Constant
`peek()`	O(1) Constant
`size()`	O(1) Constant
`isEmpty()`	O(1) Constant
`add()`	

`size =` 1

```
add(5)
add(8)
remove()
```

Implementing a Queue with Linked Nodes

QUEUE ADT

State

Collection of ordered items
Count of items

Behavior

`add(item)` add item to back
`remove()` remove and return item at front
`peek()` return item at front
`size()` count of items
`isEmpty()` count is 0?

LinkedQueue<E>

State

```
Node front
Node back
size
```

Behavior

`add` – add node to back
`remove` – return and remove node at front
`peek` – return node at front
`size` – return size
`isEmpty` – return size == 0

Big-Oh Analysis

`remove()`	O(1) Constant
`peek()`	O(1) Constant
`size()`	O(1) Constant
`isEmpty()`	O(1) Constant
`add()`	O(1) Constant

`size =` 1

```
add(5)
add(8)
remove()
```

سوال؟

Implementing a Queue with an Array (v1)

QUEUE ADT

State

Collection of ordered items
Count of items

Behavior

`add(item)` add item to back
`remove()` remove and return item at front
`peek()` return item at front
`size()` count of items
`isEmpty()` count is 0?

ArrayQueueV1<E>

State

`data[]`
`size`

Behavior

`add` – data[size] = value, if out of room grow
`remove` – return/remove at 0, shift everything
`peek` – return node at 0
`size` – return size
`isEmpty` – return size == 0

Implementing a Queue with an Array (v1)

QUEUE ADT

State

Collection of ordered items
Count of items

Behavior

add(item) add item to back
remove() remove and return item at front
peek() return item at front
size() count of items
isEmpty() count is 0?

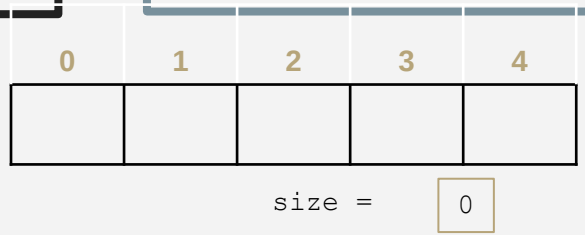
ArrayQueueV1<E>

State

data[]
size

Behavior

add – data[size] = value, if out of room grow
remove – return/remove at 0, shift everything
peek – return node at 0
size – return size
isEmpty – return size == 0



size = 0

Implementing a Queue with an Array (v1)

QUEUE ADT

State

Collection of ordered items
Count of items

Behavior

`add(item)` add item to back
`remove()` remove and return item at front
`peek()` return item at front
`size()` count of items
`isEmpty()` count is 0?

ArrayQueueV1<E>

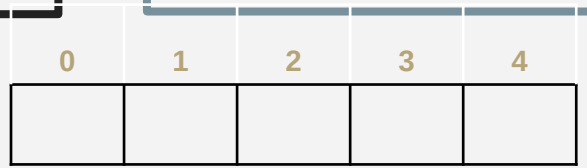
State

`data[]`
`size`

Behavior

`add` – data[size] = value, if out of room grow
`remove` – return/remove at 0, shift everything
`peek` – return node at 0
`size` – return size
`isEmpty` – return size == 0

`add(5)`



size = 0

Implementing a Queue with an Array (v1)

QUEUE ADT

State

Collection of ordered items
Count of items

Behavior

`add(item)` add item to back
`remove()` remove and return item at front
`peek()` return item at front
`size()` count of items
`isEmpty()` count is 0?

ArrayQueueV1<E>

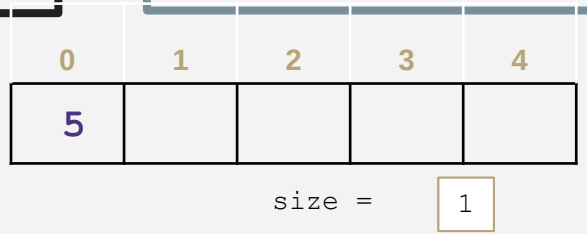
State

`data[]`
`size`

Behavior

`add` – data[size] = value, if out of room grow
`remove` – return/remove at 0, shift everything
`peek` – return node at 0
`size` – return size
`isEmpty` – return size == 0

`add(5)`



Implementing a Queue with an Array (v1)

QUEUE ADT

State

Collection of ordered items
Count of items

Behavior

`add(item)` add item to back
`remove()` remove and return item at front
`peek()` return item at front
`size()` count of items
`isEmpty()` count is 0?

ArrayQueueV1<E>

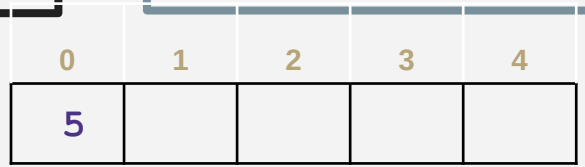
State

`data[]`
`size`

Behavior

`add` – data[size] = value, if out of room grow
`remove` – return/remove at 0, shift everything
`peek` – return node at 0
`size` – return size
`isEmpty` – return size == 0

```
add(5)
add(8)
```



size = 1

Implementing a Queue with an Array (v1)

QUEUE ADT

State

Collection of ordered items
Count of items

Behavior

`add(item)` add item to back
`remove()` remove and return item at front
`peek()` return item at front
`size()` count of items
`isEmpty()` count is 0?

ArrayQueueV1<E>

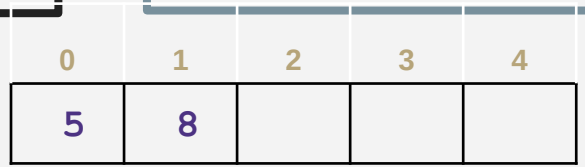
State

`data[]`
`size`

Behavior

`add` – data[size] = value, if out of room grow
`remove` – return/remove at 0, shift everything
`peek` – return node at 0
`size` – return size
`isEmpty` – return size == 0

```
add(5)
add(8)
```



size = 2

Implementing a Queue with an Array (v1)

QUEUE ADT

State

Collection of ordered items
Count of items

Behavior

`add(item)` add item to back
`remove()` remove and return item at front
`peek()` return item at front
`size()` count of items
`isEmpty()` count is 0?

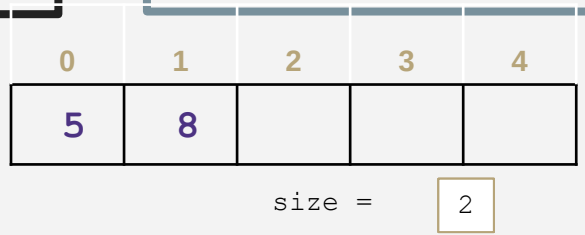
ArrayQueueV1<E>

State

`data[]`
`size`

Behavior

`add` – data[size] = value, if out of room grow
`remove` – return/remove at 0, shift everything
`peek` – return node at 0
`size` – return size
`isEmpty` – return size == 0




```
add(5)
add(8)
add(9)
```

Implementing a Queue with an Array (v1)

QUEUE ADT

State

Collection of ordered items
Count of items

Behavior

`add(item)` add item to back
`remove()` remove and return item at front
`peek()` return item at front
`size()` count of items
`isEmpty()` count is 0?

ArrayQueueV1<E>

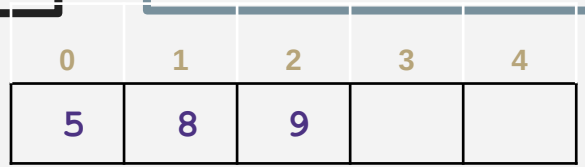
State

`data[]`
`size`

Behavior

`add` – data[size] = value, if out of room grow
`remove` – return/remove at 0, shift everything
`peek` – return node at 0
`size` – return size
`isEmpty` – return size == 0

```
add(5)
add(8)
add(9)
```



size = 3

Implementing a Queue with an Array (v1)

QUEUE ADT

State

Collection of ordered items
Count of items

Behavior

`add(item)` add item to back
`remove()` remove and return item at front
`peek()` return item at front
`size()` count of items
`isEmpty()` count is 0?

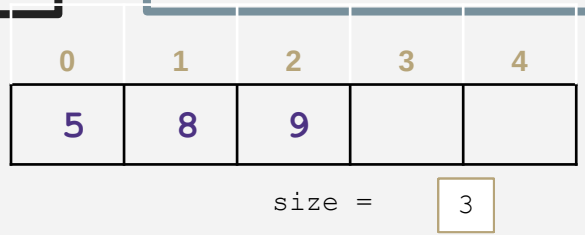
ArrayQueueV1<E>

State

`data[]`
`size`

Behavior

`add` – data[size] = value, if out of room grow
`remove` – return/remove at 0, shift everything
`peek` – return node at 0
`size` – return size
`isEmpty` – return size == 0




```
add(5)
add(8)
add(9)
remove()
```

Implementing a Queue with an Array (v1)

QUEUE ADT

State

Collection of ordered items
Count of items

Behavior

add(item) add item to back
remove() remove and return item at front
peek() return item at front
size() count of items
isEmpty() count is 0?

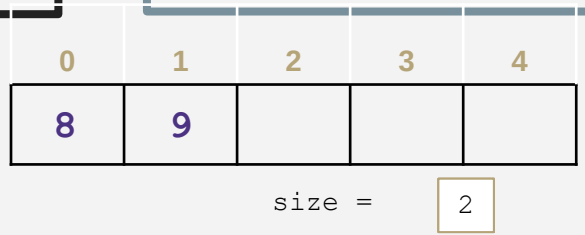
ArrayQueueV1<E>

State

data[]
size

Behavior

add – data[size] = value, if out of room grow
remove – return/remove at 0, shift everything
peek – return node at 0
size – return size
isEmpty – return size == 0




```
add(5)
add(8)
add(9)
remove()
```

Implementing a Queue with an Array (v1)

QUEUE ADT

State

Collection of ordered items
Count of items

Behavior

`add(item)` add item to back
`remove()` remove and return item at front
`peek()` return item at front
`size()` count of items
`isEmpty()` count is 0?

ArrayQueueV1<E>

State

`data[]`
`size`

Behavior

`add` – data[size] = value, if out of room grow
`remove` – return/remove at 0, shift everything
`peek` – return node at 0
`size` – return size
`isEmpty` – return size == 0

Big-Oh Analysis

`peek()`

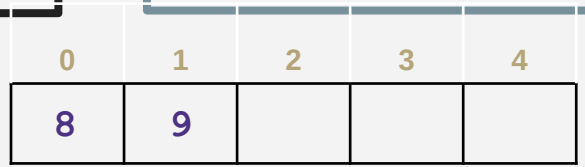
`size()`

`isEmpty()`

`add()`

`remove()`

```
add(5)
add(8)
add(9)
remove()
```



size = 2

Implementing a Queue with an Array (v1)

QUEUE ADT

State

Collection of ordered items
Count of items

Behavior

`add(item)` add item to back
`remove()` remove and return item at front
`peek()` return item at front
`size()` count of items
`isEmpty()` count is 0?

ArrayQueueV1<E>

State

`data[]`
`size`

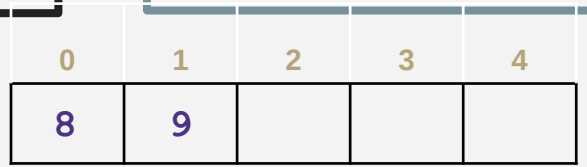
Behavior

`add` – data[size] = value, if out of room grow
`remove` – return/remove at 0, shift everything
`peek` – return node at 0
`size` – return size
`isEmpty` – return size == 0

Big-Oh Analysis

`peek()`	O(1) Constant
`size()`	
`isEmpty()`	
`add()`	
`remove()`	

```
add(5)
add(8)
add(9)
remove()
```



Implementing a Queue with an Array (v1)

QUEUE ADT

State

Collection of ordered items
Count of items

Behavior

`add(item)` add item to back
`remove()` remove and return item at front
`peek()` return item at front
`size()` count of items
`isEmpty()` count is 0?

ArrayQueueV1<E>

State

`data[]`
`size`

Behavior

`add` – data[size] = value, if out of room grow
`remove` – return/remove at 0, shift everything
`peek` – return node at 0
`size` – return size
`isEmpty` – return size == 0

0	1	2	3	4
8	9			

size = 2

```
add(5)
add(8)
add(9)
remove()
```

Big-Oh Analysis

`peek()`	O(1) Constant
`size()`	O(1) Constant
`isEmpty()`	
`add()`	
`remove()`	

Implementing a Queue with an Array (v1)

QUEUE ADT

State

Collection of ordered items
Count of items

Behavior

`add(item)` add item to back
`remove()` remove and return item at front
`peek()` return item at front
`size()` count of items
`isEmpty()` count is 0?

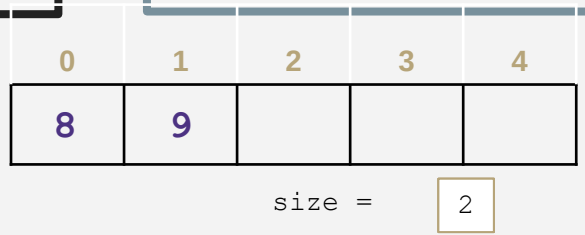
ArrayQueueV1<E>

State

`data[]`
`size`

Behavior

`add` – data[size] = value, if out of room grow
`remove` – return/remove at 0, shift everything
`peek` – return node at 0
`size` – return size
`isEmpty` – return size == 0




```
add(5)
add(8)
add(9)
remove()
```

Big-Oh Analysis

`peek()`	O(1) Constant
`size()`	O(1) Constant
`isEmpty()`	O(1) Constant
`add()`	
`remove()`	

Implementing a Queue with an Array (v1)

QUEUE ADT

State

Collection of ordered items
Count of items

Behavior

`add(item)` add item to back
`remove()` remove and return item at front
`peek()` return item at front
`size()` count of items
`isEmpty()` count is 0?

ArrayQueueV1<E>

State

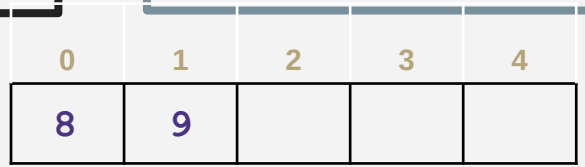
`data[]`
`size`

Behavior

`add` – data[size] = value, if out of room grow
`remove` – return/remove at 0, shift everything
`peek` – return node at 0
`size` – return size
`isEmpty` – return size == 0

Big-Oh Analysis

`peek()`	O(1) Constant
`size()`	O(1) Constant
`isEmpty()`	O(1) Constant
`add()`	
`remove()`	




```
add(5)
add(8)
add(9)
remove()
```

What do you think the worst possible runtime of `add()` & `remove()` could be?

Implementing a Queue with an Array (v1)

QUEUE ADT

State

Collection of ordered items
Count of items

Behavior

`add(item)` add item to back
`remove()` remove and return item at front
`peek()` return item at front
`size()` count of items
`isEmpty()` count is 0?

ArrayQueueV1<E>

State

`data[]`
`size`

Behavior

`add` – data[size] = value, if out of room grow
`remove` – return/remove at 0, shift everything
`peek` – return node at 0
`size` – return size
`isEmpty` – return size == 0

0	1	2	3	4
8	9			

size = 2

```
add(5)
add(8)
add(9)
remove()
```

Big-Oh Analysis

`peek()`	O(1) Constant
`size()`	O(1) Constant
`isEmpty()`	O(1) Constant
`add()`	O(n) Linear: if we need to resize O(1) Constant: otherwise
`remove()`	

Implementing a Queue with an Array (v1)

QUEUE ADT

State

Collection of ordered items
Count of items

Behavior

`add(item)` add item to back
`remove()` remove and return item at front
`peek()` return item at front
`size()` count of items
`isEmpty()` count is 0?

ArrayQueueV1<E>

State

`data[]`
`size`

Behavior

`add` – data[size] = value, if out of room grow
`remove` – return/remove at 0, shift everything
`peek` – return node at 0
`size` – return size
`isEmpty` – return size == 0

0	1	2	3	4
8	9			

size = 2

```
add(5)
add(8)
add(9)
remove()
```

Big-Oh Analysis

`peek()`	O(1) Constant
`size()`	O(1) Constant
`isEmpty()`	O(1) Constant
`add()`	O(n) Linear: if we need to resize O(1) Constant: otherwise
`remove()`	O(n) Linear

سوال؟

Consider Data Structure Invariants

- **Invariant**: a property of a data structure that is always true between operations
  - true when finishing any operation, so it can be counted on to be true when starting an operation.

Consider Data Structure Invariants

- **Invariant**: a property of a data structure that is always true between operations
  - true when finishing any operation, so it can be counted on to be true when starting an operation.
- ArrayQueueV1 is basically an `ArrayList`. What invariants does `ArrayList` have for its `data` array?

Consider Data Structure Invariants

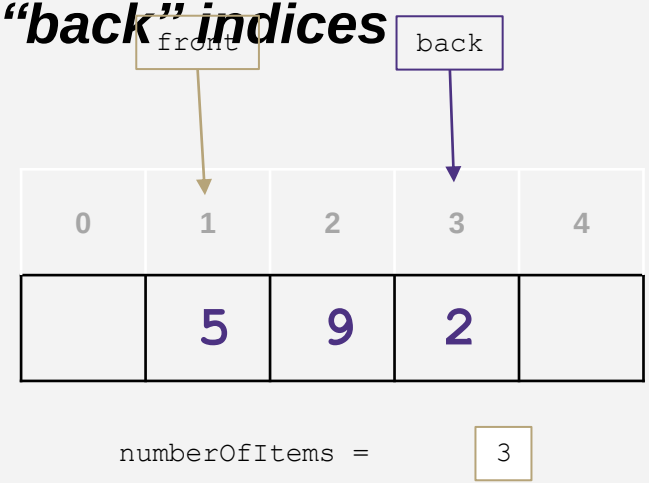
- **Invariant**: a property of a data structure that is always true between operations
  - true when finishing any operation, so it can be counted on to be true when starting an operation.
- ArrayQueueV1 is basically an `ArrayList`. What invariants does `ArrayList` have for its `data` array?
  - The i-th item in the list is stored in data[i]

Consider Data Structure Invariants

- **Invariant**: a property of a data structure that is always true between operations
  - true when finishing any operation, so it can be counted on to be true when starting an operation.
- ArrayQueueV1 is basically an `ArrayList`. What invariants does `ArrayList` have for its `data` array?
  - The i-th item in the list is stored in data[i]
    - Notice: serving this invariant is what slows down the operation. Could we choose a different invariant?

Implementing a Queue with an Array

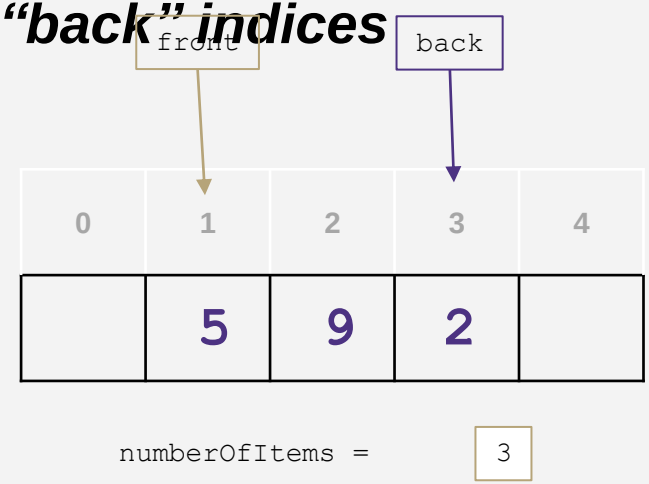
Wrapping Around with "front" and "back" indices



Implementing a Queue with an Array

Wrapping Around with "front" and "back" indices

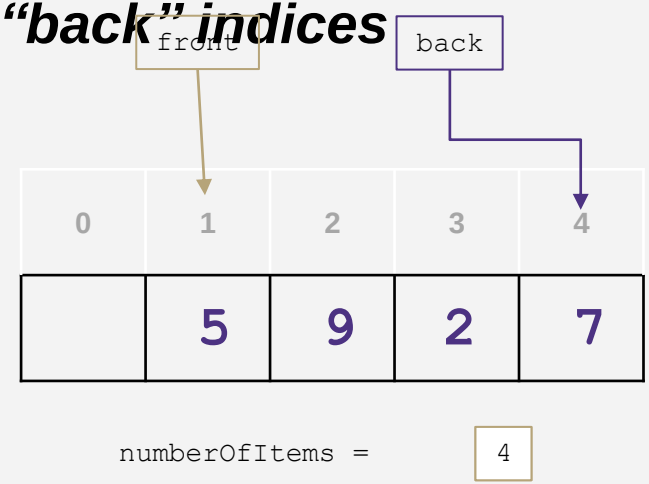
add(7)



Implementing a Queue with an Array

Wrapping Around with "front" and "back" indices

add(7)

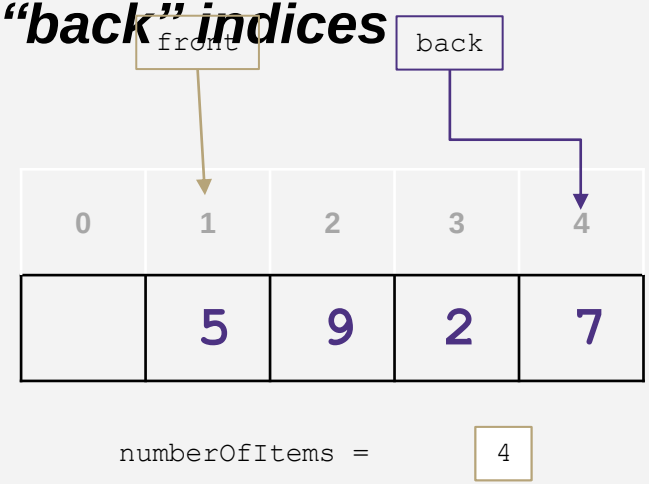


Implementing a Queue with an Array

Wrapping Around with "front" and "back" indices

add(7)

add(4)

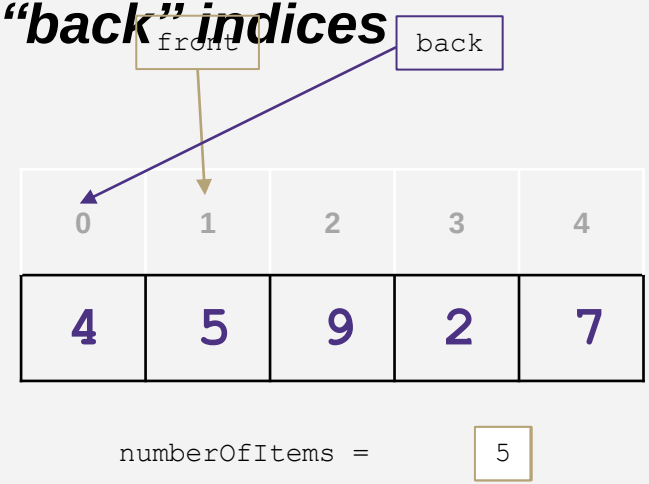


Implementing a Queue with an Array

Wrapping Around with "front" and "back" indices

add(7)

add(4)



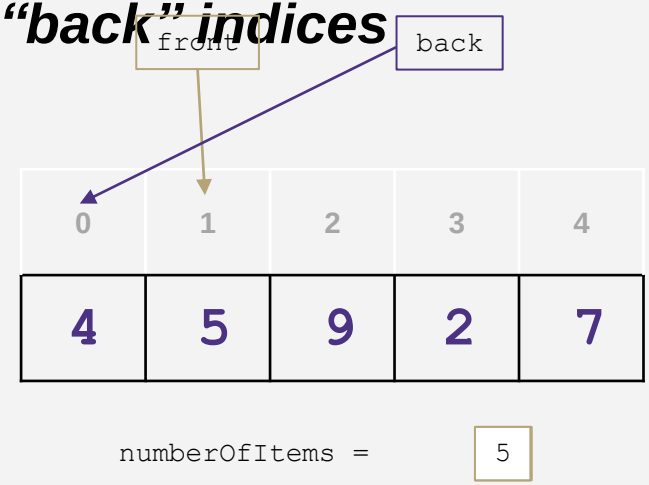
Implementing a Queue with an Array

Wrapping Around with "front" and "back" indices

add(7)

add(4)

add(1)



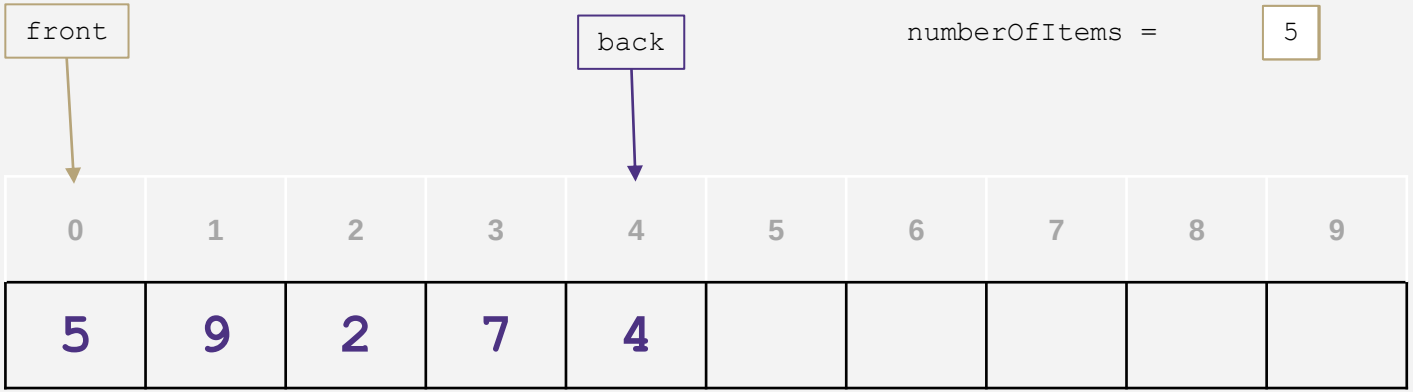
Implementing a Queue with an Array

Wrapping Around with "front" and "back" indices

add(7)

add(4)

add(1)



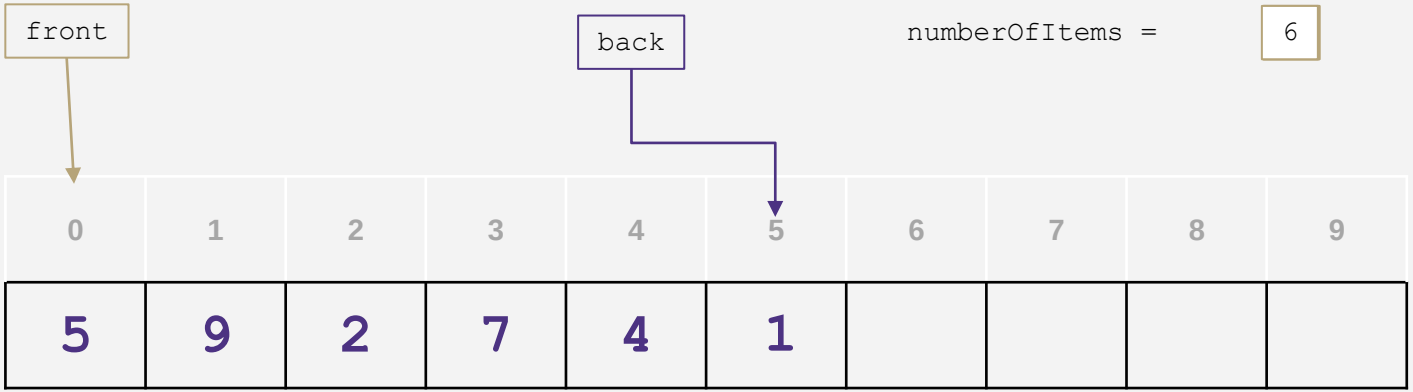
Implementing a Queue with an Array

Wrapping Around with "front" and "back" indices

add(7)

add(4)

add(1)



Implementing a Queue with an Array

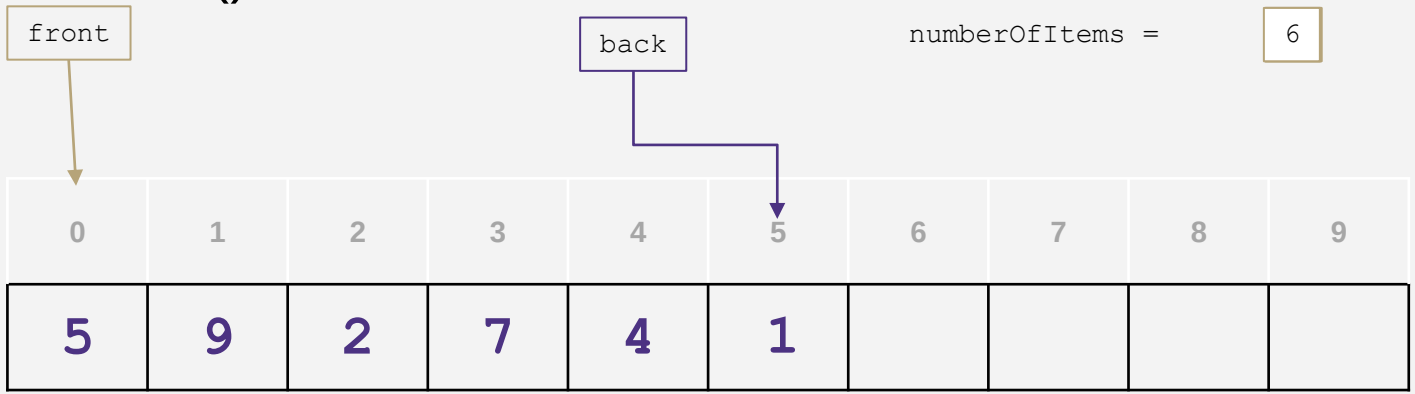
Wrapping Around with "front" and "back" indices

add(7)

add(4)

add(1)

remove()



Implementing a Queue with an Array

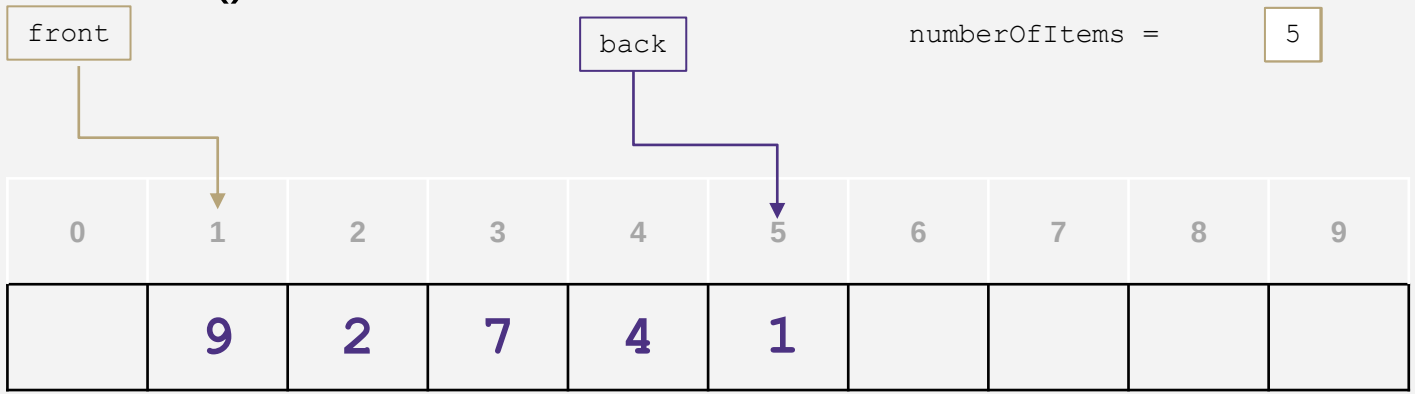
Wrapping Around with "front" and "back" indices

add(7)

add(4)

add(1)

remove()



Implementing a Queue with an Array (v2)

QUEUE ADT

State

Collection of ordered items
Count of items

Behavior

`add(item)` add item to back
`remove()` remove and return item at front
`peek()` return item at front
`size()` count of items
`isEmpty()` count is 0?

ArrayQueueV2<E>

State

```
data[],   front,
size,     back
```

Behavior

`add` – data[back] = value, back++, size++, if out of room grow
`remove` – return data[front], size--, front++
`peek` – return data[front]
`size` – return size
`isEmpty` – return size == 0

Implementing a Queue with an Array (v2)

QUEUE ADT

State

Collection of ordered items
Count of items

Behavior

add(item) add item to back
remove() remove and return item at front
peek() return item at front
size() count of items
isEmpty() count is 0?

ArrayQueueV2<E>

State

data[], front,
size, back

Behavior

add – data[back] = value, back++, size++, if out of room grow
remove – return data[front], size--, front++
peek – return data[front]
size – return size
isEmpty – return size == 0

Big-Oh Analysis

peek()

size()

isEmpty()

add()

remove()

Implementing a Queue with an Array (v2)

QUEUE ADT

State

Collection of ordered items
Count of items

Behavior

`add(item)` add item to back
`remove()` remove and return item at front
`peek()` return item at front
`size()` count of items
`isEmpty()` count is 0?

ArrayQueueV2<E>

State

`data[], front, size, back`

Behavior

`add` – data[back] = value, back++, size++, if out of room grow
`remove` – return data[front], size--, front++
`peek` – return data[front]
`size` – return size
`isEmpty` – return size == 0

Big-Oh Analysis

`peek()`	O(1) Constant
`size()`	
`isEmpty()`	
`add()`	
`remove()`	

Implementing a Queue with an Array (v2)

QUEUE ADT

State

Collection of ordered items
Count of items

Behavior

add(item) add item to back
remove() remove and return item at front
peek() return item at front
size() count of items
isEmpty() count is 0?

ArrayQueueV2<E>

State

data[], front,
size, back

Behavior

add – data[back] = value, back++, size++, if out of room grow
remove – return data[front], size--, front++
peek – return data[front]
size – return size
isEmpty – return size == 0

Big-Oh Analysis

peek()	O(1) Constant
size()	O(1) Constant
isEmpty()	
add()	
remove()	

Implementing a Queue with an Array (v2)

QUEUE ADT

State

Collection of ordered items
Count of items

Behavior

`add(item)` add item to back
`remove()` remove and return item at front
`peek()` return item at front
`size()` count of items
`isEmpty()` count is 0?

ArrayQueueV2<E>

State

`data[], front, size, back`

Behavior

`add` – data[back] = value, back++, size++, if out of room grow
`remove` – return data[front], size--, front++
`peek` – return data[front]
`size` – return size
`isEmpty` – return size == 0

Big-Oh Analysis

`peek()`	O(1) Constant
`size()`	O(1) Constant
`isEmpty()`	O(1) Constant
`add()`	
`remove()`	

Implementing a Queue with an Array (v2)

QUEUE ADT

State

Collection of ordered items
Count of items

Behavior

`add(item)` add item to back
`remove()` remove and return item at front
`peek()` return item at front
`size()` count of items
`isEmpty()` count is 0?

ArrayQueueV2<E>

State

`data[], front, size, back`

Behavior

`add` – data[back] = value, back++, size++, if out of room grow
`remove` – return data[front], size--, front++
`peek` – return data[front]
`size` – return size
`isEmpty` – return size == 0

Big-Oh Analysis

`peek()`	O(1) Constant
`size()`	O(1) Constant
`isEmpty()`	O(1) Constant
`add()`	O(n) Linear: if we need to resize O(1) Constant: otherwise
`remove()`	

Implementing a Queue with an Array (v2)

QUEUE ADT

State

Collection of ordered items
Count of items

Behavior

`add(item)` add item to back
`remove()` remove and return item at front
`peek()` return item at front
`size()` count of items
`isEmpty()` count is 0?

ArrayQueueV2<E>

State

`data[], front, size, back`

Behavior

`add` – data[back] = value, back++, size++, if out of room grow
`remove` – return data[front], size--, front++
`peek` – return data[front]
`size` – return size
`isEmpty` – return size == 0

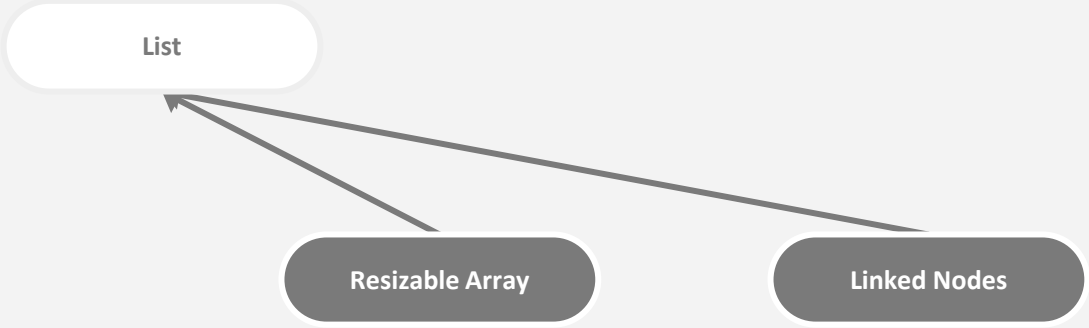
Big-Oh Analysis

`peek()`	O(1) Constant
`size()`	O(1) Constant
`isEmpty()`	O(1) Constant
`add()`	O(n) Linear: if we need to resize O(1) Constant: otherwise
`remove()`	O(1) Constant

سوال؟

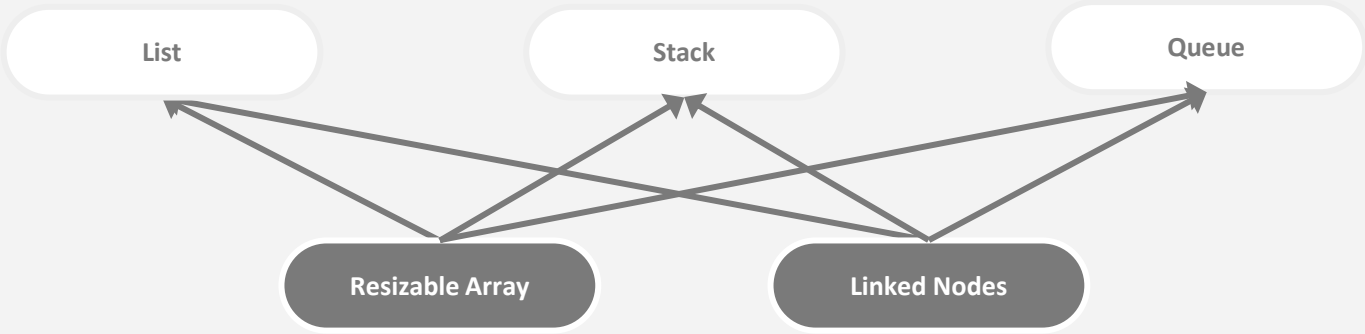
ADTs & Data Structures

- We’ve now seen that just like an ADT can be implemented by multiple data structures, a data structure can implement multiple ADTs



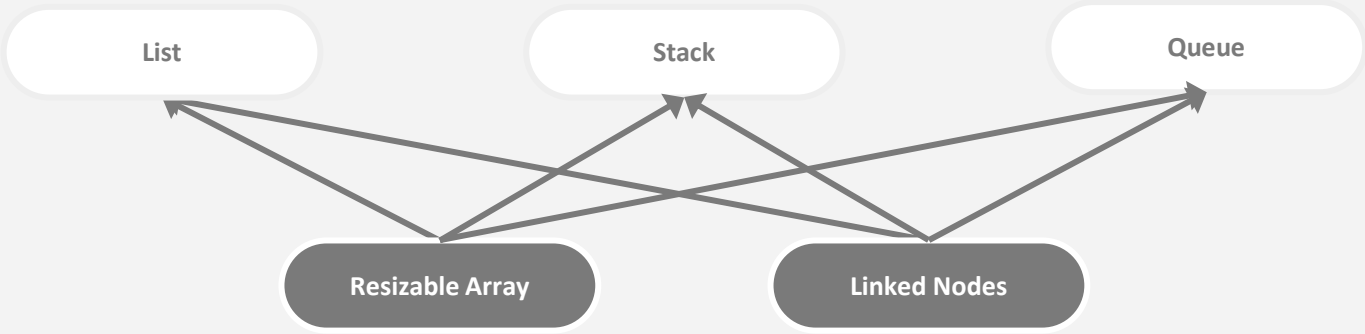
ADTs & Data Structures

- We’ve now seen that just like an ADT can be implemented by multiple data structures, a data structure can implement multiple ADTs



ADTs & Data Structures

- We've now seen that just like an ADT can be implemented by multiple data structures, a data structure can implement multiple ADTs



- But the ADT decides how it can be used
  - An ArrayList used as a List should support `get()`, but when used as a Stack should not

سوال؟