COMP2402
Abstract Data Types and Algorithms

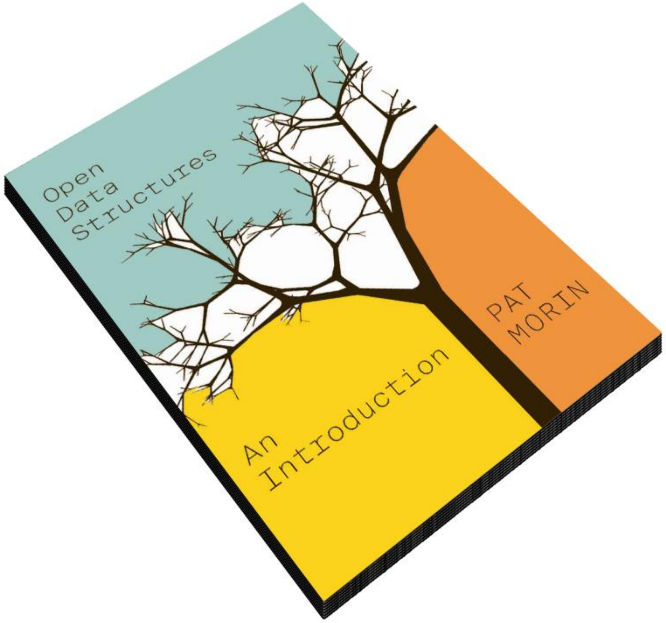
Array-Backed Queues and Deques

Reading Assignment

Open Data Structures in Java

by Pat Morin

Chapter 2.2, 2.3, 2.4



Fundamental and Supporting Operations

the Abstract Data Type known as the "Queue"
Guarantees the following Fundamental Operations:

`add(o)` *alternatively:* *`enqueue()`*

Insert the Object o at the Back of the Queue

`remove()` *alternatively:* *`dequeue()`*

Remove and Return the Object from the Front of the Queue

Fundamental and Supporting Operations

the Abstract Data Type known as the "Queue"
Often Has the following Supporting Operations:

`size()`

Return the Number of Elements in the Queue

`element()`

Return (Without Removing) the Object from the Front of the Queue

`resize()`

Resize the Underlying Data Structure (e.g., the Backing Array)

Array-Backed Lists as Queues

the Operations Guaranteed by the List Interface Include All of the Operations for the Stack, Queue, and Deque

Does the Array-Backed List Discussed Previously* Support the Functionality of a Stack? a Queue?

*(*ArrayStack in the textbook; with add(i, o) and remove(i) methods)*

How?

i.e., How do you Implement:

push/pop using add/remove?

enqueue/dequeue using add/remove?

Array-Backed Lists as Queues

the Operations Guaranteed by the List Interface Include All of the Operations for the Stack, Queue, and Deque

Does the Array-Backed List Discussed Previously* Support the Functionality of a Stack? a Queue?

(*ArrayStack in the textbook; with add(i, o) and remove(i) methods)

How? Don't! (Why Not?)

i.e., How do you Implement:

push/pop using add/remove?

enqueue/dequeue using add/remove?

Array-Backed Lists as Queues

What are the Worst-Case Time Complexity* of:

*(*again using the methods from the ArrayStack in the textbook)*

`remove(i)`?

`add(i, o)`?

`push(o)` *...where* `push` *is actually a call to* `add`?

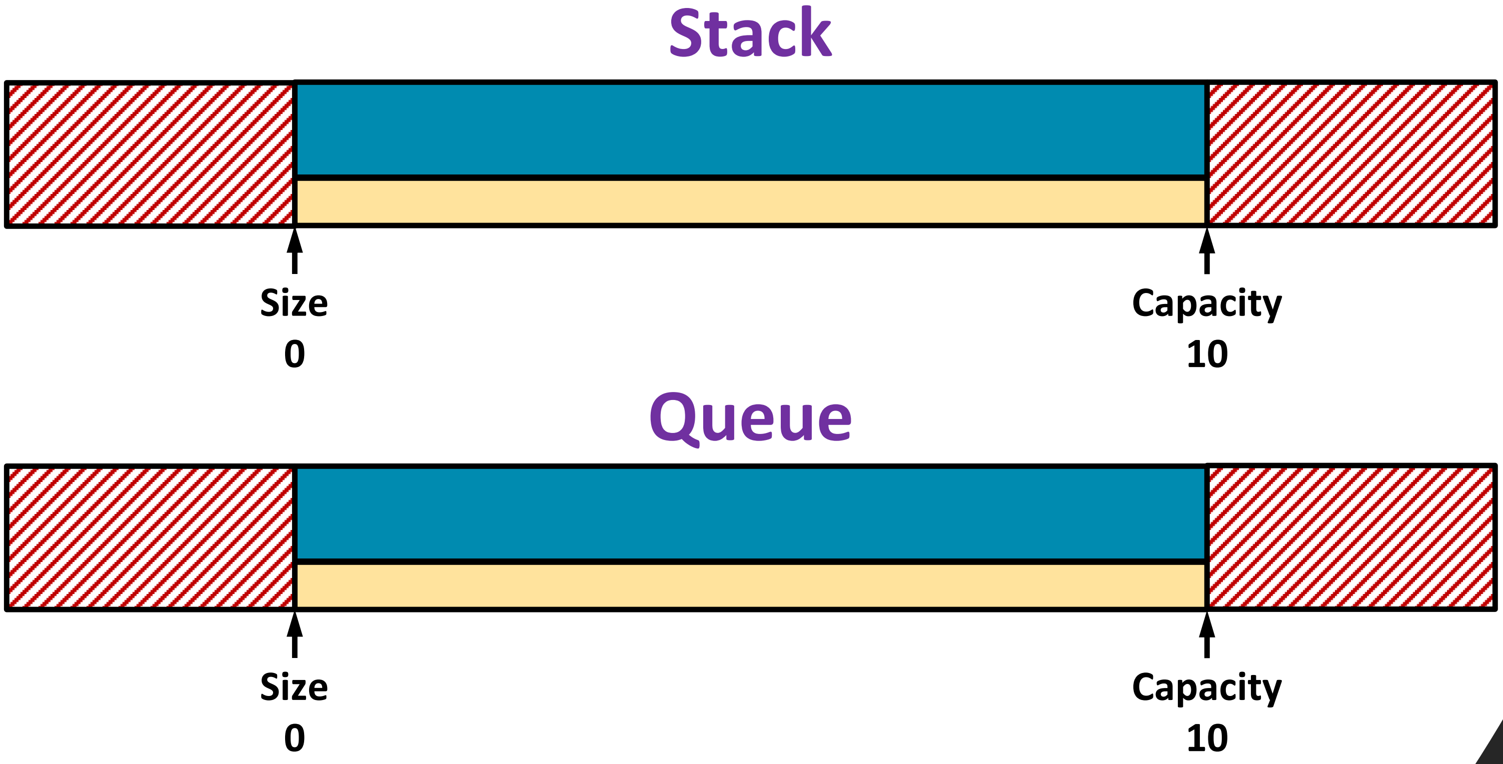
`dequeue(o)` *...where* `dequeue` *is actually a call to* `add`?

`pop()` *...where* `pop` *is actually a call to* `remove`?

`enqueue()` *...where* `enqueue` *is actually a call to* `remove`?

Overloading Naïve Stacks and Queues

Suppose you had a Stack and a Queue and Both Used an Array of Length 20 that Cannot be Resized as the Underlying Data Structure…



Stack

Size
0

Capacity
10

Queue

Size
0

Capacity
10

Overloading Naïve Stacks and Queues

the Stack has Variable top to Store the Index of the Top of the Stack, it implements push(o) by Adding o at Index top and Incrementing top, and it implements pop() by Returning the Element at Index top and Decrementing top

the Queue has Variables front and back to Store the Indices of the Front and Back of the Queue, it implements add(o) by Adding o at Index back and Incrementing back, and it implements remove() by Returning the Element at Index front and then Incrementing front

Overloading Naïve Stacks and Queues

Now Suppose you begin Inserting and Deleting Elements (via push/pop or add/remove) according to the Patterns:

1. insert, delete, insert, delete, insert, delete... etc.
2. insert, delete, insert, insert, delete, delete... etc.

How Many Operations (i.e., Inserts or Deletes) can Occur Before an IndexOutOfBounds Exception...

... for the Stack?

... for the Queue?

"Repairing" the Array-Backed Queue

If Size (i.e., Number of Elements, Not Capacity) is Not Increasing or Decreasing Over Time there should be No Need to Resize, Regardless of the Number of Operations

this is Accomplished by "Wrapping" the Queue Within the Array (as long as the Queue Size is Within the Capacity)

Circular Array-Backed Queue

Instead of an Implementation of Queue where the Occupied Portion of the Backing Array is:

data[front]		*data[front]*
data[front+1]	→	*data[front+1]*
...	equivalently	...
data[front+back]		*data[front+size-1]*

```
enqueue(o):
      add o at data[back]
      increment back (and/or size)

dequeue():
      remove from data[front]
      increment front
```

Circular Array-Backed Queue

Use an Implementation
where the Occupied Portion of the Backing Array is:

```
data[front%data.length]
data[(front+1)%data.length]
...
data[(front+size-1)%data.length]
```

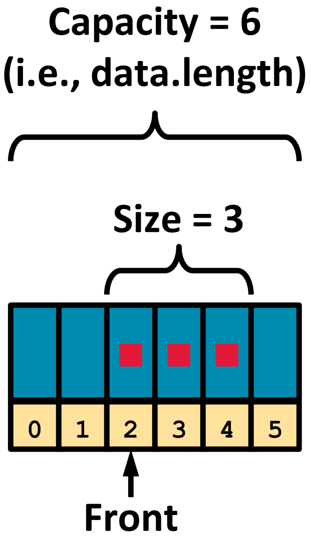
```
enqueu(o):
      add o at data[(front+size)%data.length]
      increment size

dequeue():
      remove from data[front]
      front ← (front+1)%data.length
      // i.e., increment front with wrap
```

Circular Array-Backed Queue

e.g.,

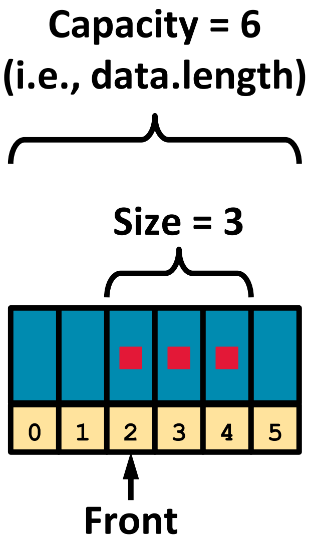
(adapted from the textbook)



Circular Array-Backed Queue

e.g.,

(adapted from the textbook)


Capacity = 6
(i.e., data.length)

Size = 3

0	1	2	3	4	5
		🟥	🟥	🟥	

Front

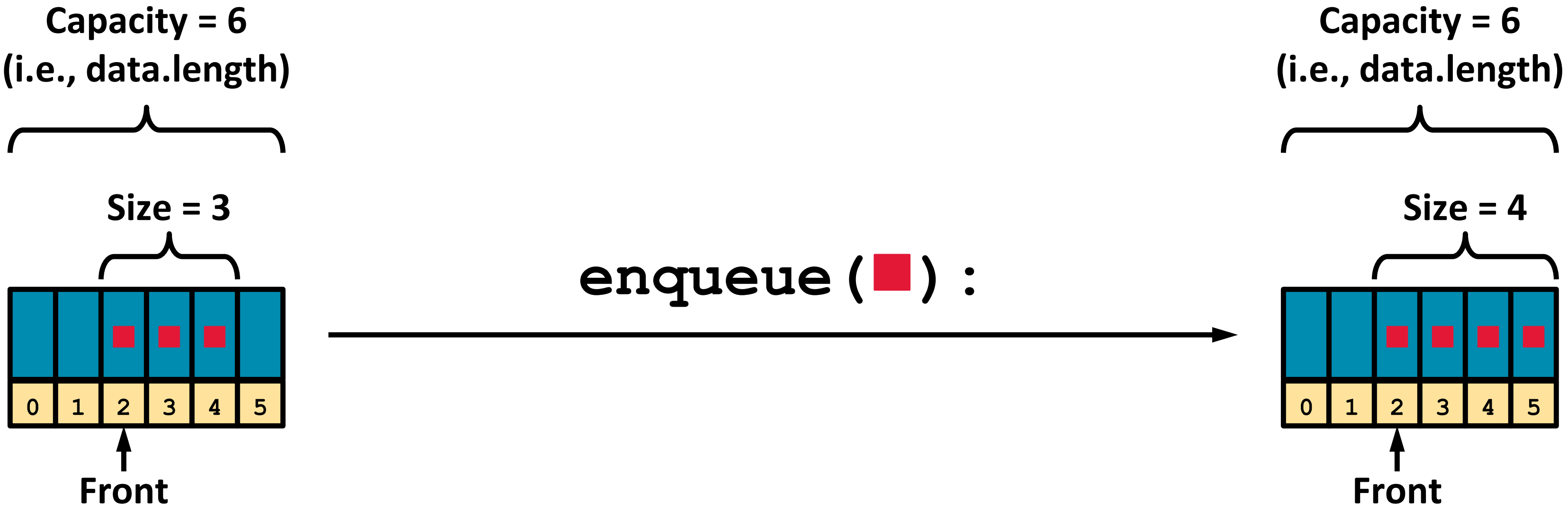
`enqueue(🟥):`

```
data[(front+size)%data.length] ← 🟥
data[(2+3)%6] ← 🟥
data[5] ← 🟥
size ← 4
```

Circular Array-Backed Queue

e.g.,

(adapted from the textbook)



Capacity = 6
(i.e., data.length)

Size = 3

Front

enqueue(🟥):

Capacity = 6
(i.e., data.length)

Size = 4

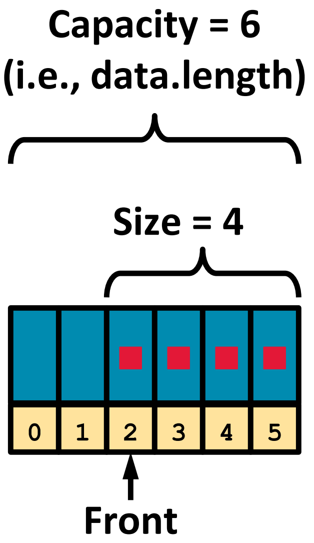
Front

```
data[(front+size)%data.length] ← 🟥
data[(2+3)%6] ← 🟥
data[5] ← 🟥
size ← 4
```

Circular Array-Backed Queue

e.g.,

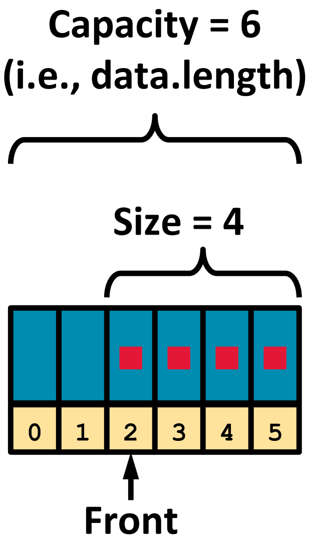
(adapted from the textbook)



Circular Array-Backed Queue

e.g.,

(adapted from the textbook)



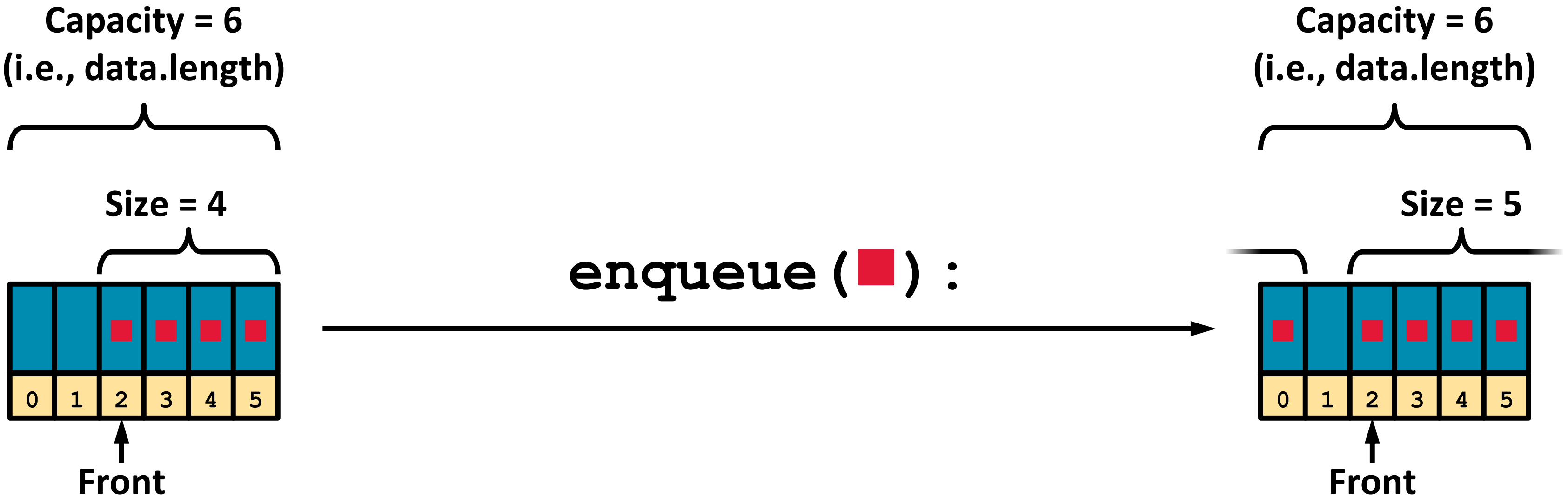
```
enqueue(🟥):

data[(front+size)%data.length] ← 🟥
data[(2+4)%6] ← 🟥
data[0] ← 🟥
size ← 5
```

Circular Array-Backed Queue

e.g.,

(adapted from the textbook)



Capacity = 6
(i.e., data.length)

Size = 4

enqueue(🟥):

Capacity = 6
(i.e., data.length)

Size = 5

```
data[(front+size)%data.length] ← 🟥
data[(2+4)%6] ← 🟥
data[0] ← 🟥
size ← 5
```

Fundamental and Supporting Operations

the Abstract Data Type known as the "Deque" Guarantees the following Fundamental Operations:

`addFirst(o)`

Insert the Object o at the Front of the Deque

`removeFirst()`

Remove and Return the Object from the Front of the Deque

`addLast(o)`

Insert the Object o at the Back of the Deque

`removeLast()`

Remove and Return the Object from the Back of the Deque

Stacks, Queues, and Deques

the Deque Uses the Same "Circular Array" Technique as seen in the Previous Implementation of Queue

now imagine Using Stacks, Queues, and Deques as Lists:

i.e., Implement `add(i, o)` and `remove(i)` Using Only `push`/`pop`, `enqueue`/`dequeue`, and `addFirst`/`removeFirst`/`addLast`/`removeLast`

What are Good Scenarios (i.e., Values for `i`)?

Stacks, Queues, and Deques

If a Deque-Based Implementation of List should Perform `add(i, o)` and `remove(i)` Quickly with `i` Near Size, then In What Direction should we Shift Elements?

Stacks, Queues, and Deques

If a Deque-Based Implementation of List should Perform `add(i, o)` and `remove(i)` Quickly with `i` Near Size, then In What Direction should we Shift Elements?

If the Index into which we are Inserting is Near the Front, Shift the Elements Left (with Wrapping)

If the Index into which we are Inserting is Near the Back, Shift the Elements Right (with Wrapping)

Stacks, Queues, and Deques

If a Deque-Based Implementation of List should Perform `add(i, o)` and `remove(i)` Quickly with `i` Near Size, then In What Direction should we Shift Elements?

If the Index into which we are Inserting is Near the Front, Shift the Elements Left (with Wrapping)

If the Index into which we are Inserting is Near the Back, Shift the Elements Right (with Wrapping)

Don't Forget that Resizing the Array will still be Necessary if the Size of the Stack/Queue/Deque Exceeds the Capacity of the Backing Array