Abstract Data Types

EECS 214, Fall 2018

What is an ADT?

An ADT defines:

- A set of (abstract) values
- A set of (abstract) operations on those values

What is an ADT?

An ADT defines:

- A set of (abstract) values
- A set of (abstract) operations on those values

An ADT omits:

- How the values are concretely represented
- How the operations work

ADT: Stack

Looks like: |3 4 5⟩

ADT: Stack

Looks like: $|3\ 4\ 5\rangle$

Signature:

- *push*(Stack, Element): Void
- *pop*(Stack): Element
- *empty?*(Stack): Bool

ADT: Stack

Looks like: |3 4 5⟩

Signature:

```
interface STACK:
    def push(self, element)
    def pop(self)
    def empty?(self)
```

ADT: Stack

Looks like: |3 4 5⟩

Signature:

```
interface STACK[T]:
    def push(self, element: T) -> VoidC
    def pop(self) -> T
    def empty?(self) -> bool?
```

ADT: Queue (FIFO)

Looks like: $\langle 3\ 4\ 5 \langle$

ADT: Queue (FIFO)

Looks like: ⟨3 4 5⟨

```
interface QUEUE[T]:
    def enqueue(self, element: T) -> VoidC
    def dequeue(self) -> T
    def empty?(self) -> bool?
```

Stack versus Queue

```
interface STACK[T]:
    def push(self, element: T) -> VoidC
    def pop(self) -> T
    def empty?(self) -> bool?

interface QUEUE[T]:
    def enqueue(self, element: T) -> VoidC
    def dequeue(self) -> T
    def empty?(self) -> bool?
```

Adding laws

$$\{p\} \quad f(x) \Rightarrow y \quad \{q\}$$

means that if precondition p is true when we apply f to x then we will get y as a result, and postcondition q will be true afterward.

Adding laws

$$\{p\} \quad f(x) \Rightarrow y \quad \{q\}$$

means that if precondition p is true when we apply f to x then we will get y as a result, and postcondition q will be true afterward.

Examples:

$$\{a = [2, 4, 6, 8]\} \quad a[2] \Rightarrow 6 \quad \{a = [2, 4, 6, 8]\}$$

$$\{a = [2, 4, 6, 8]\} \quad a[2] = 0 \quad \{a = [2, 4, 0, 8]\}$$

ADT: Stack

Looks like: $|3\ 4\ 5\rangle$

Signature:

```
interface STACK[T]:
    def push(self, element: T) -> VoidC
    def pop(self) -> T
    def empty?(self) -> bool?
```

Laws:

$$|\rangle.\mathit{empty}?() \Rightarrow \top$$

$$|e_1 \ldots e_k e_{k+1}\rangle.\mathit{empty}?() \Rightarrow \bot$$

$$\{s = |e_1 \ldots e_k\rangle\}\ s.\mathit{push}(e)\ \{s = |e_1 \ldots e_k e\rangle\}$$

$$\{s = |e_1 \ldots e_k e_{k+1}\rangle\}\ s.\mathit{pop}() \Rightarrow e_{k+1}\ \{s = |e_1 \ldots e_k\rangle\}$$

ADT: Queue (FIFO)

Looks like: $\langle 3\ 4\ 5 \langle$

Signature:

```
interface QUEUE[T]:
    def enqueue(self, element: T) -> VoidC
    def dequeue(self) -> T
    def empty?(self) -> bool?
```

Laws:

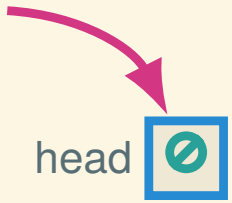
$$\langle \langle .\mathit{empty}?() \Rightarrow \top$$

$$\langle e_1 \ldots e_k e_{k+1} \langle .\mathit{empty}?() \Rightarrow \bot$$

$$\{q = \langle e_1 \ldots e_k \langle\}\ q.\mathit{enqueue}(e)\ \{q = \langle e_1 \ldots e_k e \langle\}$$

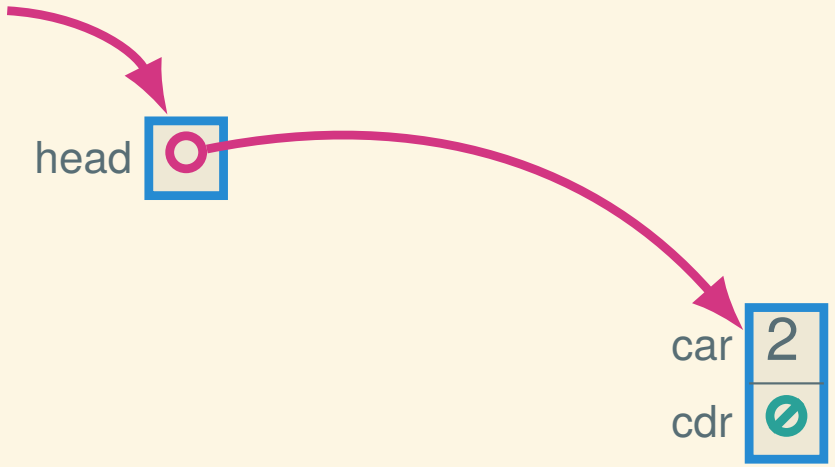
$$\{q = \langle e_1 e_2 \ldots e_k \langle\}\ q.\mathit{dequeue}() \Rightarrow e_1\ \{q = \langle e_2 \ldots e_k \langle\}$$

Stack implementation: linked list



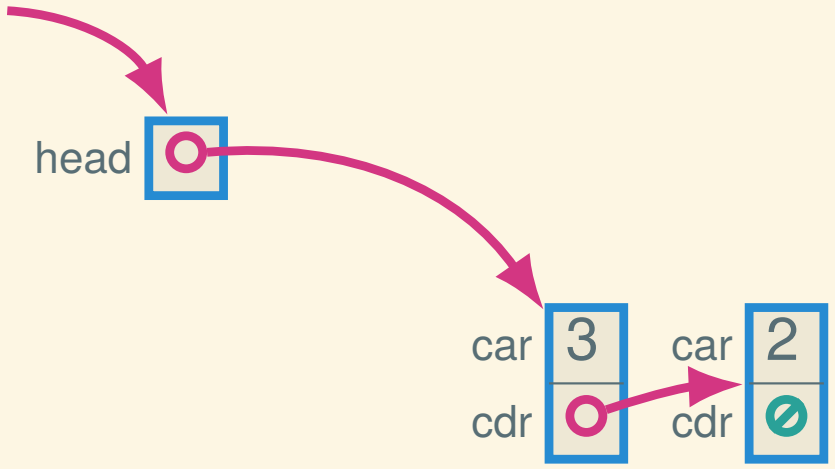
let s = ListStack()

Stack implementation: linked list



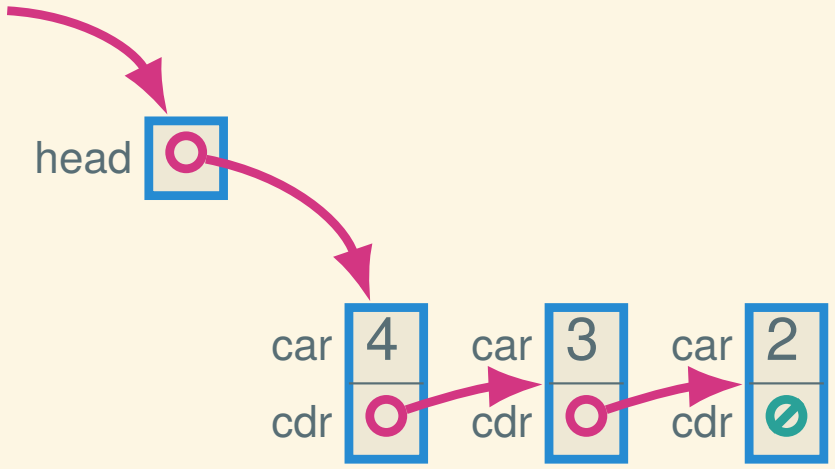

```
let s = ListStack()
s.push(2)
```

Stack implementation: linked list



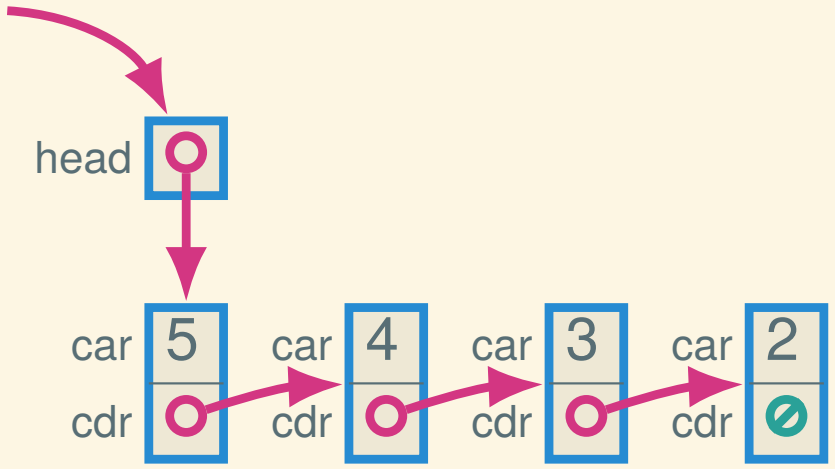

```
let s = ListStack()
s.push(2)
s.push(3)
```

Stack implementation: linked list



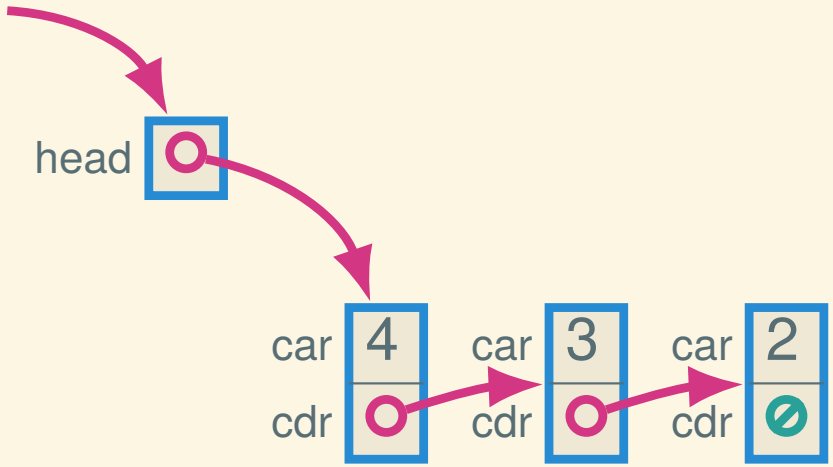

```
let s = ListStack()
s.push(2)
s.push(3)
s.push(4)
```

Stack implementation: linked list



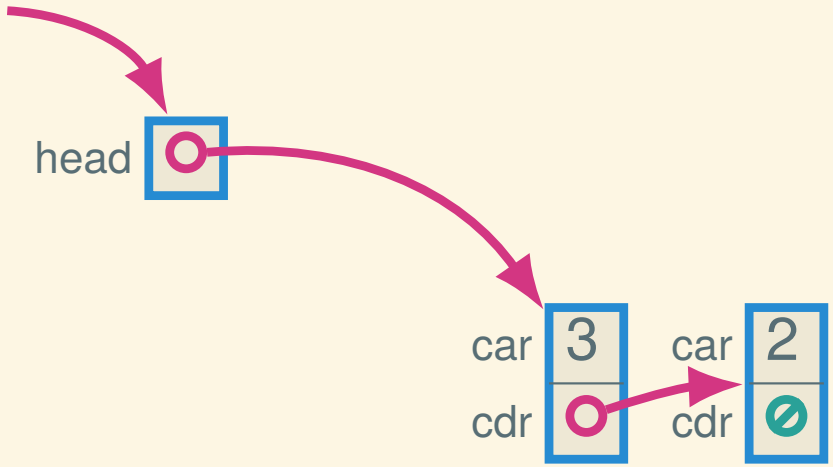

```
let s = ListStack()
s.push(2)
s.push(3)
s.push(4)
s.push(5)
```

Stack implementation: linked list



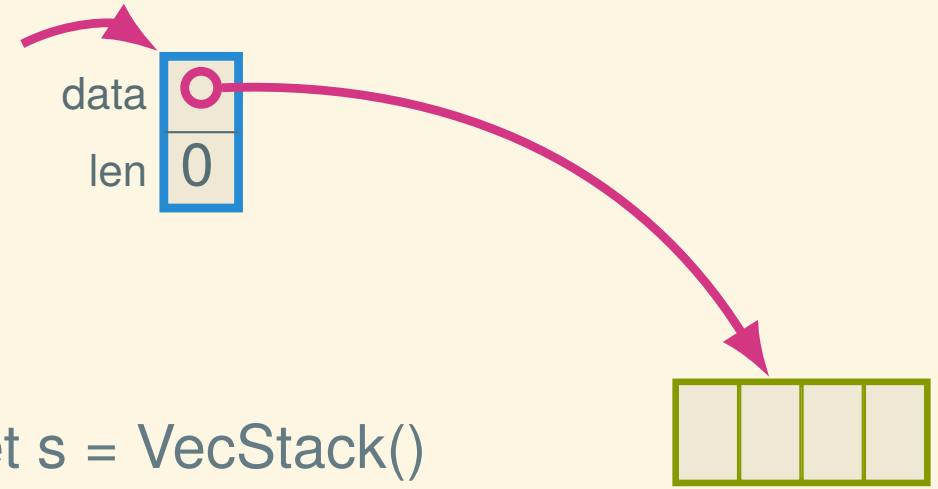

```
let s = ListStack()
s.push(2)
s.push(3)
s.push(4)
s.push(5)
s.pop()
```

Stack implementation: linked list



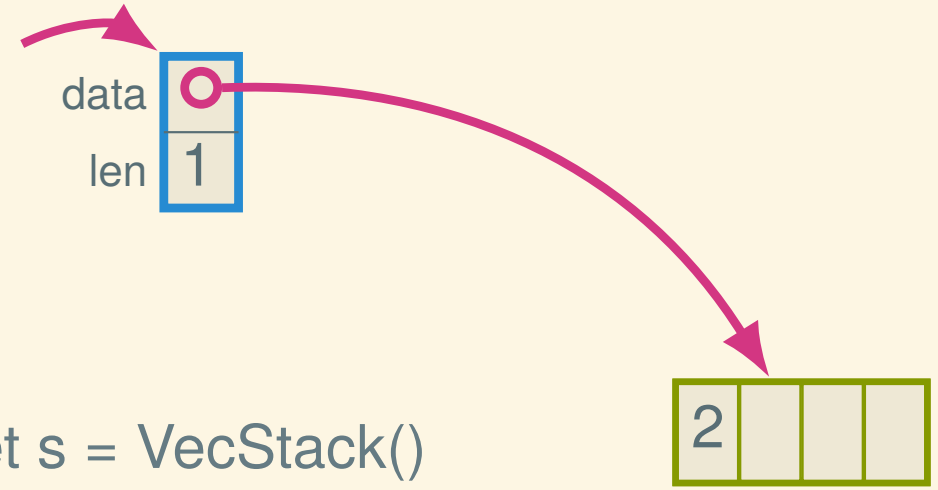

```
let s = ListStack()
s.push(2)
s.push(3)
s.push(4)
s.push(5)
s.pop()
s.pop()
```

Stack implementation: array



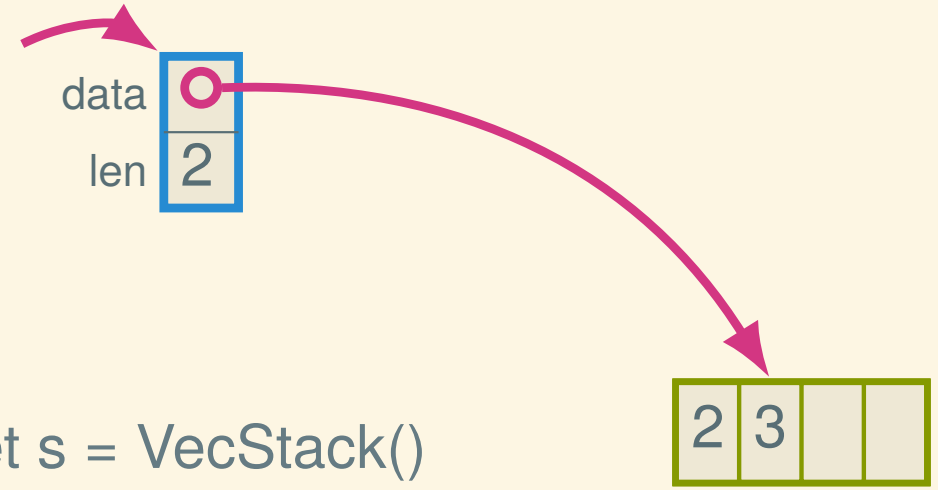
let s = VecStack()

Stack implementation: array



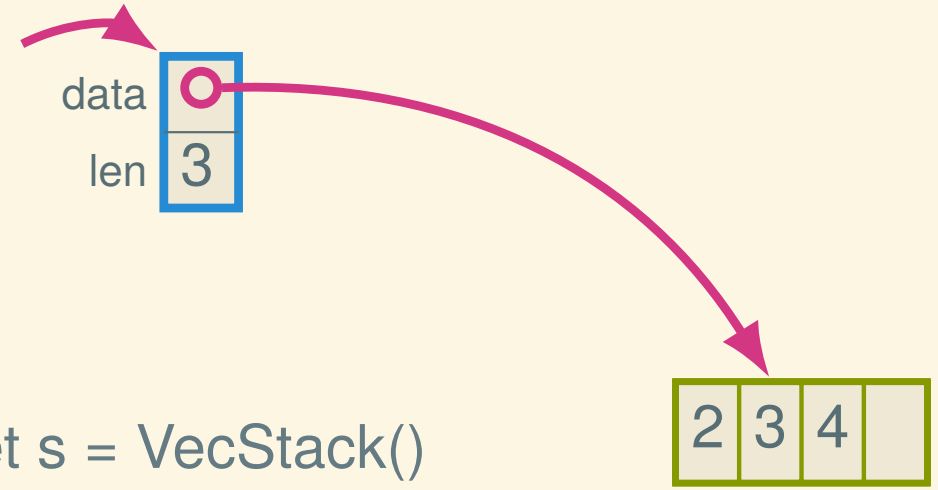

```
let s = VecStack()
s.push(2)
```

Stack implementation: array



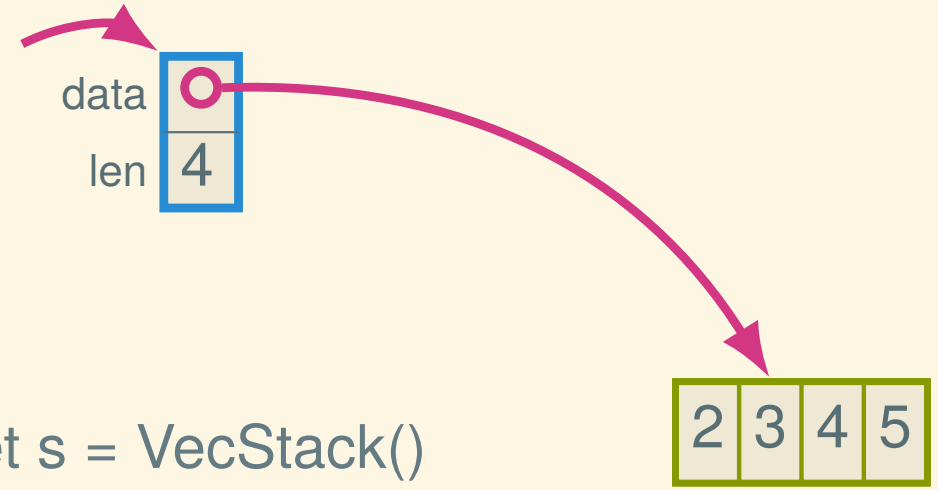

```
let s = VecStack()
s.push(2)
s.push(3)
```

Stack implementation: array



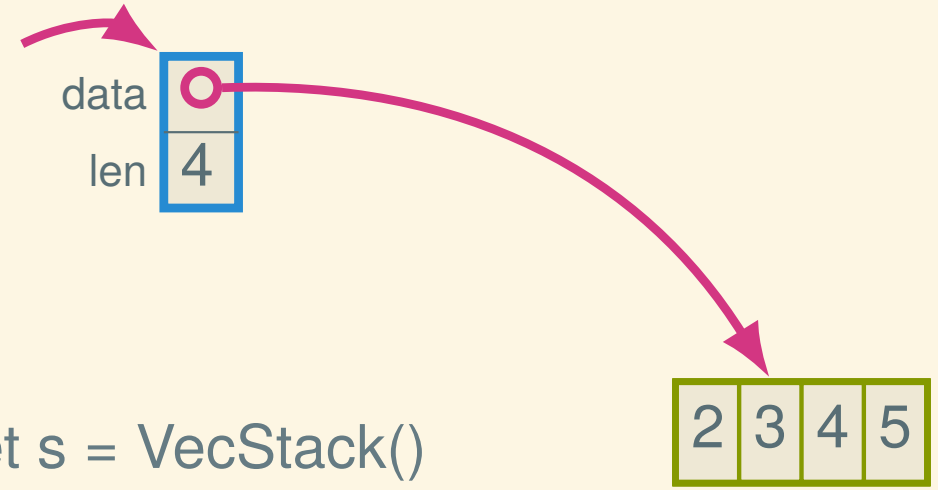

```
let s = VecStack()
s.push(2)
s.push(3)
s.push(4)
```

Stack implementation: array




```
let s = VecStack()
s.push(2)
s.push(3)
s.push(4)
s.push(5)
```

Stack implementation: array




```
let s = VecStack()
s.push(2)
s.push(3)
s.push(4)
s.push(5)
s.push(6)
```

ADT: Stack

Looks like: $|3\ 4\ 5\rangle$

Signature:

```
interface STACK[T]:
    def push(self, element: T) -> VoidC  # O(1)
    def pop(self) -> T                   # O(1)
    def empty?(self) -> bool?            # O(1)
```

Laws:

$$|\rangle.\mathit{empty}?() \Rightarrow \top$$

$$|e_1 \ldots e_k e_{k+1}\rangle.\mathit{empty}?() \Rightarrow \bot$$

$$\{s = |e_1 \ldots e_k\rangle\}\ s.\mathit{push}(e)\ \{s = |e_1 \ldots e_k e\rangle\}$$

$$\{s = |e_1 \ldots e_k e_{k+1}\rangle\}\ s.\mathit{pop}() \Rightarrow e_{k+1}\ \{s = |e_1 \ldots e_k\rangle\}$$

Trade-offs: linked list stack versus array stack

- Linked list stack only fills up when memory fills up, whereas array stack has a fixed size (or must reallocate)
- Array stack has better constant factors: cache locality and no (or rare) allocation
- Array stack space usage is tighter; linked list is smoother

ADT: Queue (FIFO)

Looks like: $\langle 3\ 4\ 5 \langle$

Signature:

```
interface QUEUE[T]:
    def enqueue(self, element: T) -> VoidC  # O(1)
    def dequeue(self) -> T                  # O(1)
    def empty?(self) -> bool?               # O(1)
```

Laws:

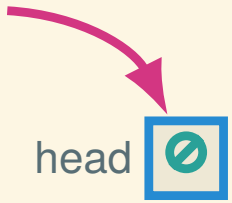
$$\langle \langle .\mathit{empty}?() \Rightarrow \top$$

$$\langle e_1 \ldots e_k e_{k+1} \langle .\mathit{empty}?() \Rightarrow \bot$$

$$\{q = \langle e_1 \ldots e_k \langle\}\ q.\mathit{enqueue}(e)\ \{q = \langle e_1 \ldots e_k e \langle\}$$

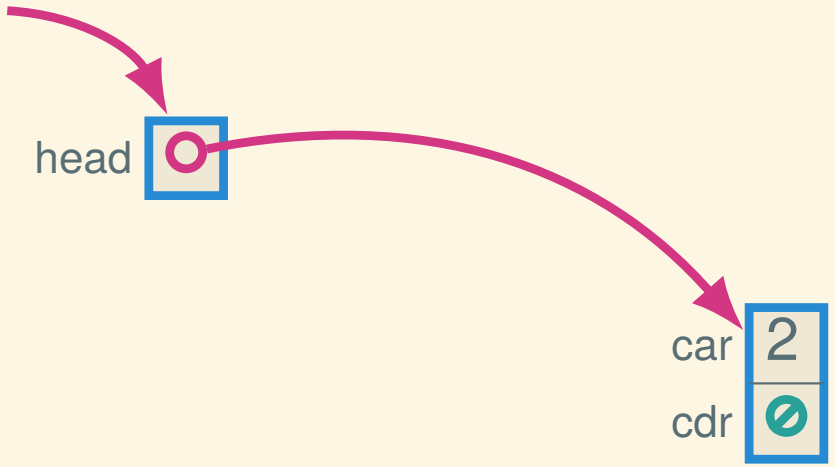
$$\{q = \langle e_1 e_2 \ldots e_k \langle\}\ q.\mathit{dequeue}() \Rightarrow e_1\ \{q = \langle e_2 \ldots e_k \langle\}$$

Queue implementation: linked list?



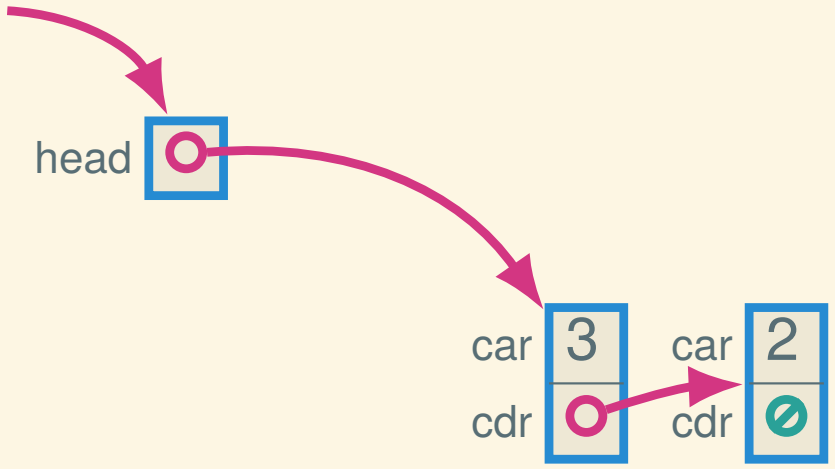
let q = LinkedListQueue()

Queue implementation: linked list?



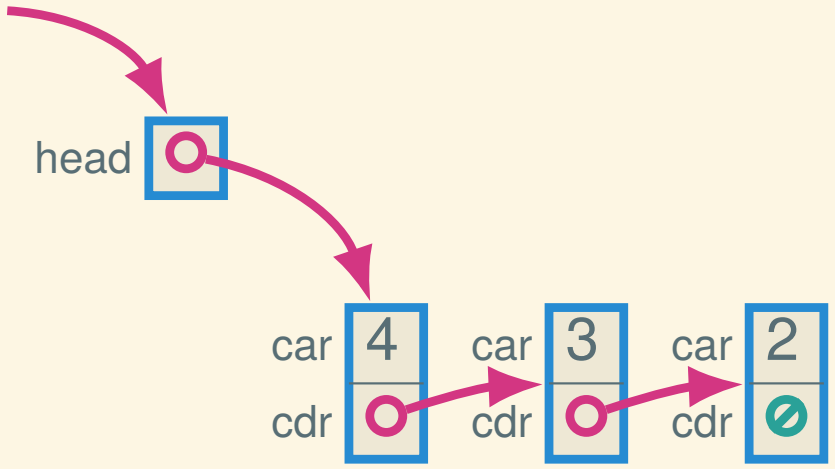

```
let q = LinkedListQueue()
q.enqueue(2)
```

Queue implementation: linked list?



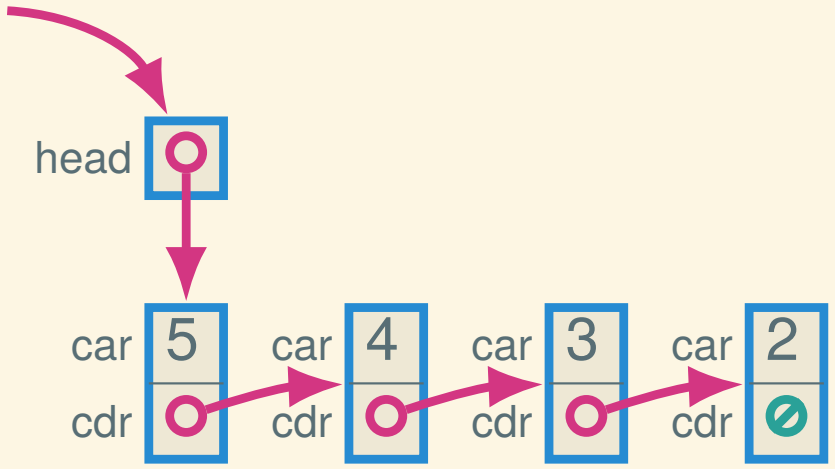

```
let q = LinkedListQueue()
q.enqueue(2)
q.enqueue(3)
```

Queue implementation: linked list?



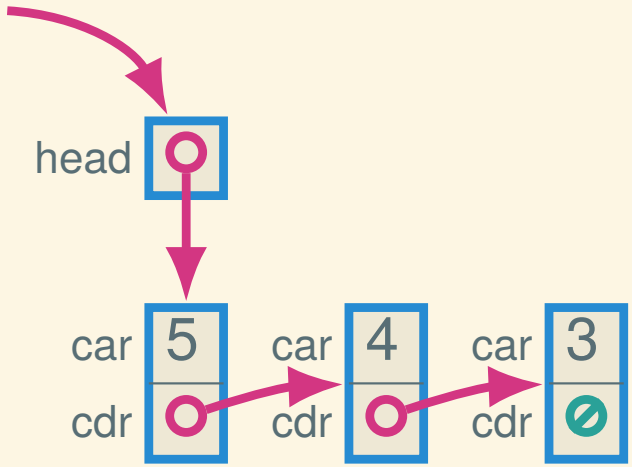

```
let q = LinkedListQueue()
q.enqueue(2)
q.enqueue(3)
q.enqueue(4)
```

Queue implementation: linked list?



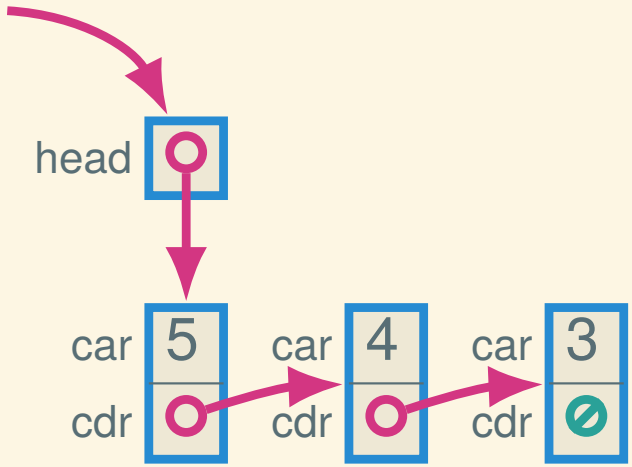

```
let q = LinkedListQueue()
q.enqueue(2)
q.enqueue(3)
q.enqueue(4)
q.enqueue(5)
```

Queue implementation: linked list?




```
let q = LinkedListQueue()
q.enqueue(2)
q.enqueue(3)
q.enqueue(4)
q.enqueue(5)
q.dequeue()
```

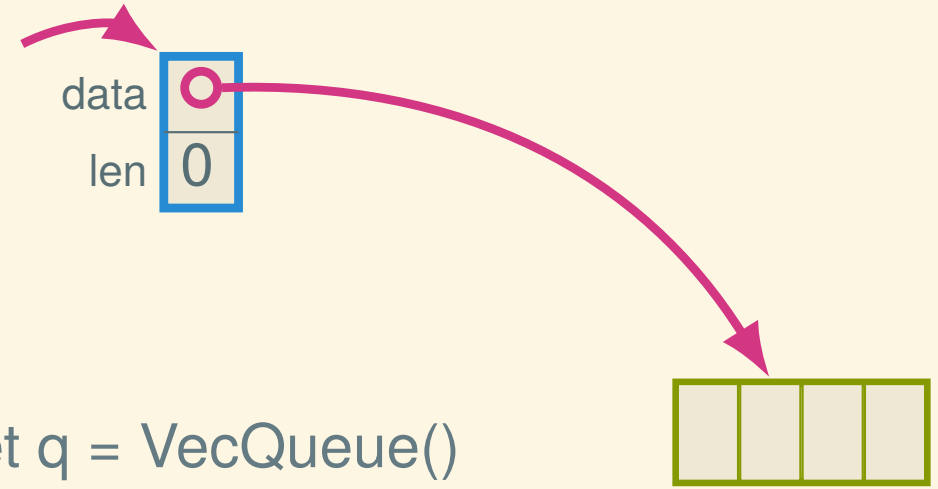
Queue implementation: linked list?




```
let q = LinkedListQueue()
q.enqueue(2)
q.enqueue(3)
q.enqueue(4)
q.enqueue(5)
```

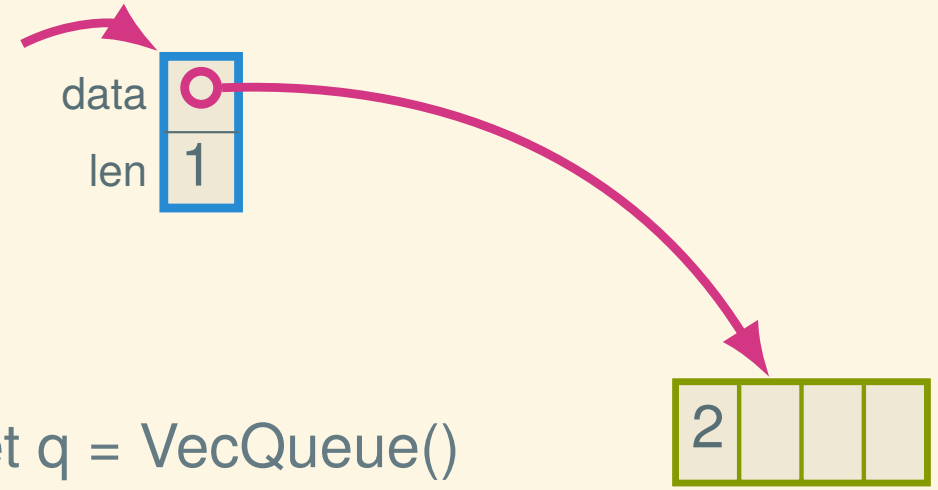
q.dequeue() — $\mathcal{O}(n)$???

Queue implementation: array?



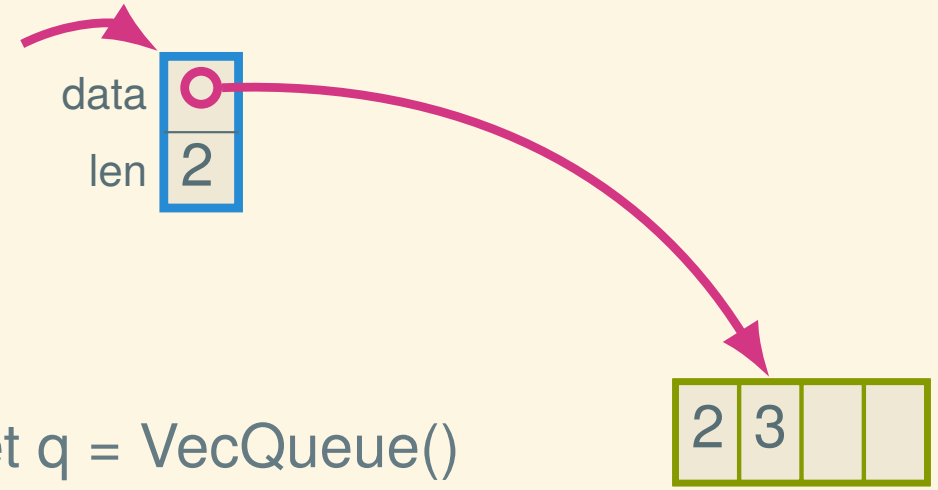
let q = VecQueue()

Queue implementation: array?



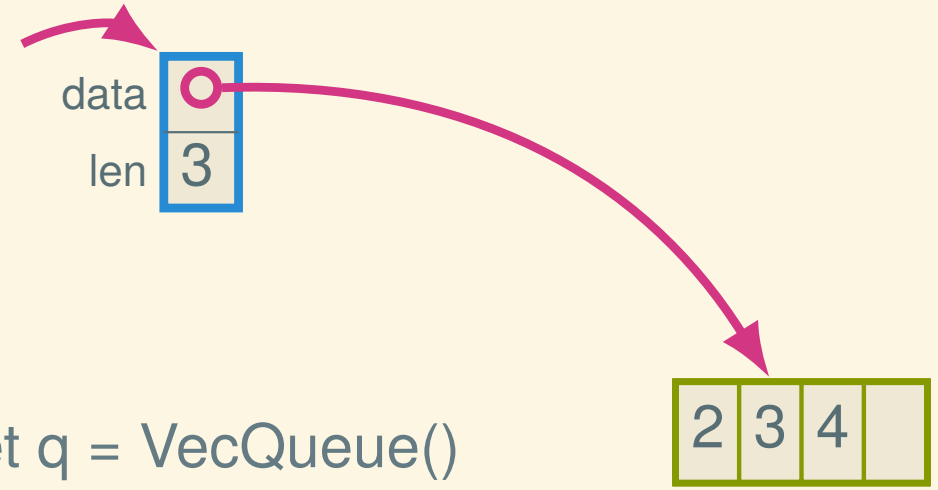

```
let q = VecQueue()
q.enqueue(2)
```

Queue implementation: array?



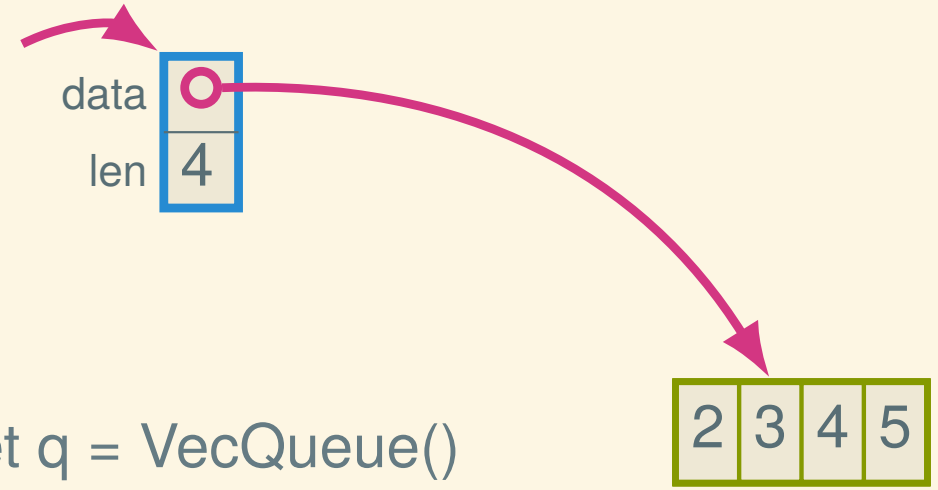

```
let q = VecQueue()
q.enqueue(2)
q.enqueue(3)
```

Queue implementation: array?



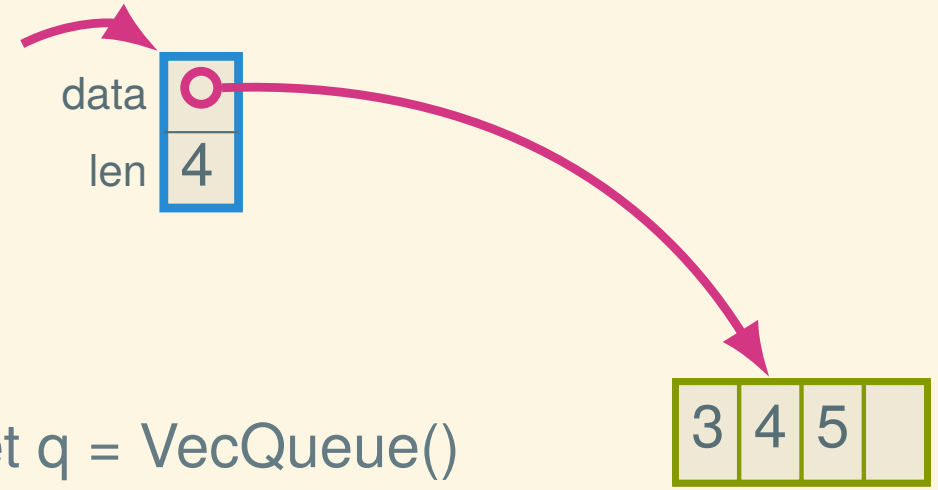

```
let q = VecQueue()
q.enqueue(2)
q.enqueue(3)
q.enqueue(4)
```

Queue implementation: array?



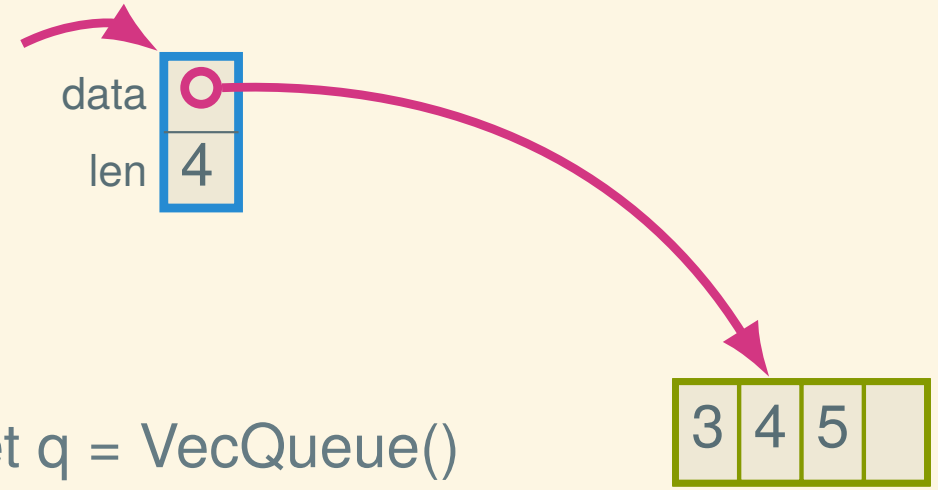

```
let q = VecQueue()
q.enqueue(2)
q.enqueue(3)
q.enqueue(4)
q.enqueue(5)
```

Queue implementation: array?



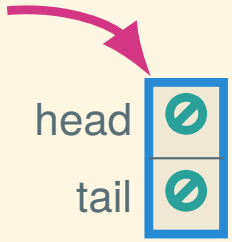

```
let q = VecQueue()
q.enqueue(2)
q.enqueue(3)
q.enqueue(4)
q.enqueue(5)
q.dequeue()
```

Queue implementation: array?



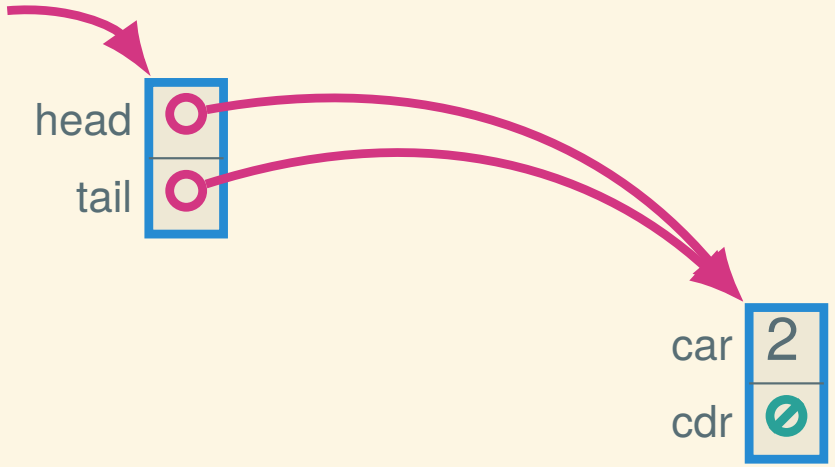
let q = VecQueue()
q.enqueue(2)
q.enqueue(3)
q.enqueue(4)
q.enqueue(5)
q.dequeue() — $\mathcal{O}(n)$???

Queue impl.: linked list with tail pointer



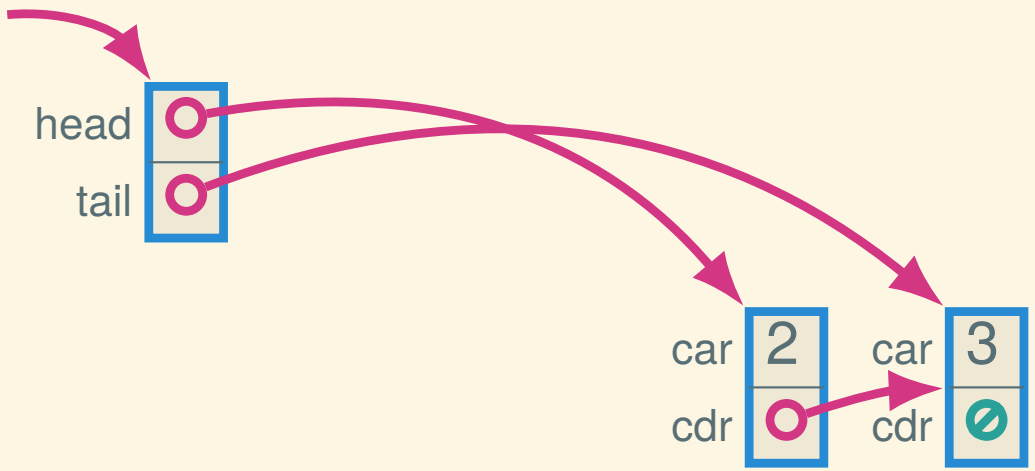
let q = LinkedListQueue()

Queue impl.: linked list with tail pointer



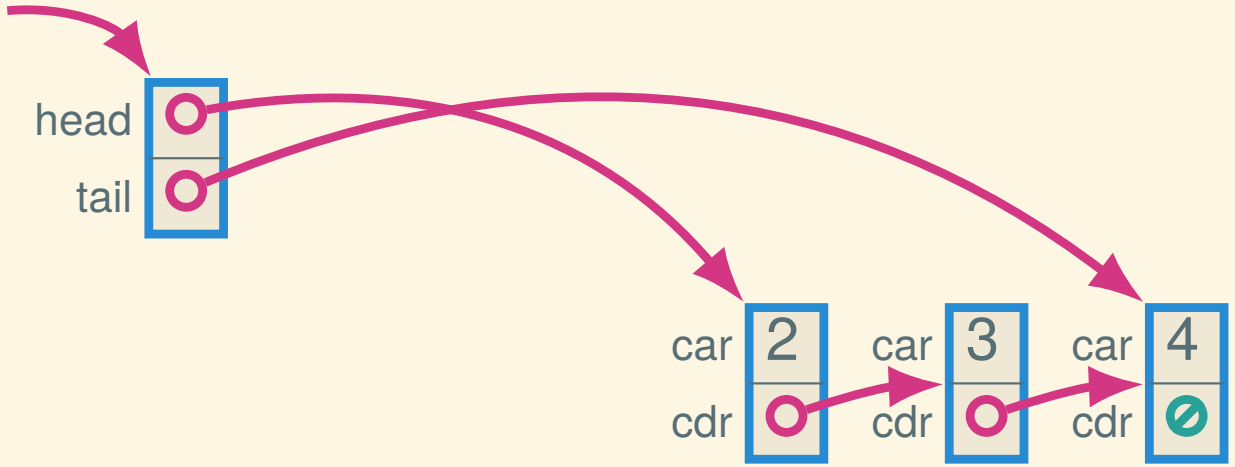
let q = LinkedListQueue()
q.enqueue(2)

Queue impl.: linked list with tail pointer



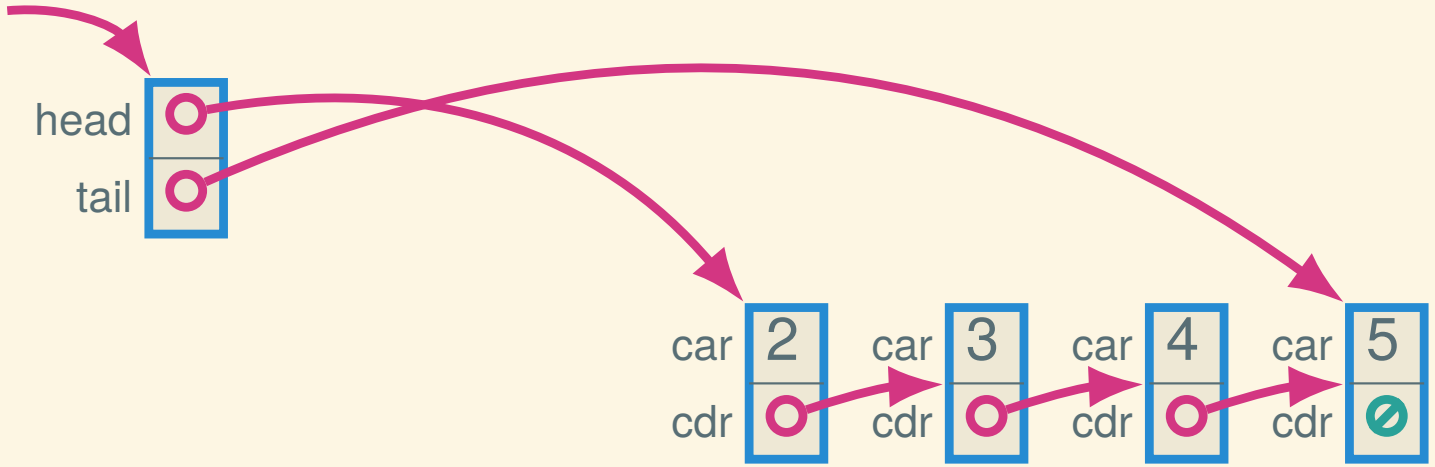

```
let q = LinkedListQueue()
q.enqueue(2)
q.enqueue(3)
```

Queue impl.: linked list with tail pointer



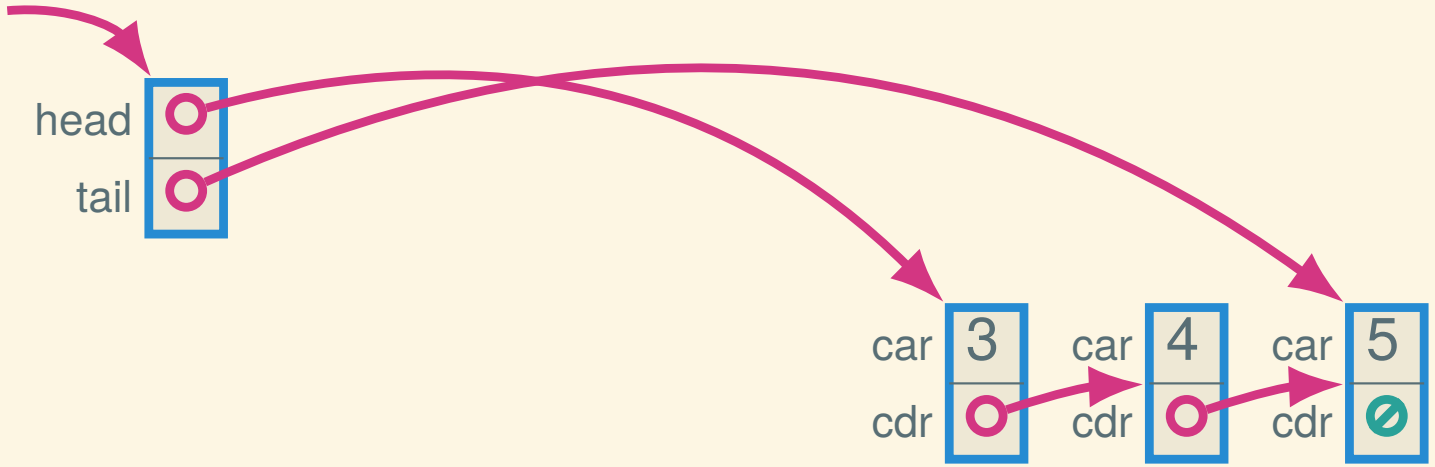

```
let q = LinkedListQueue()
q.enqueue(2)
q.enqueue(3)
q.enqueue(4)
```

Queue impl.: linked list with tail pointer



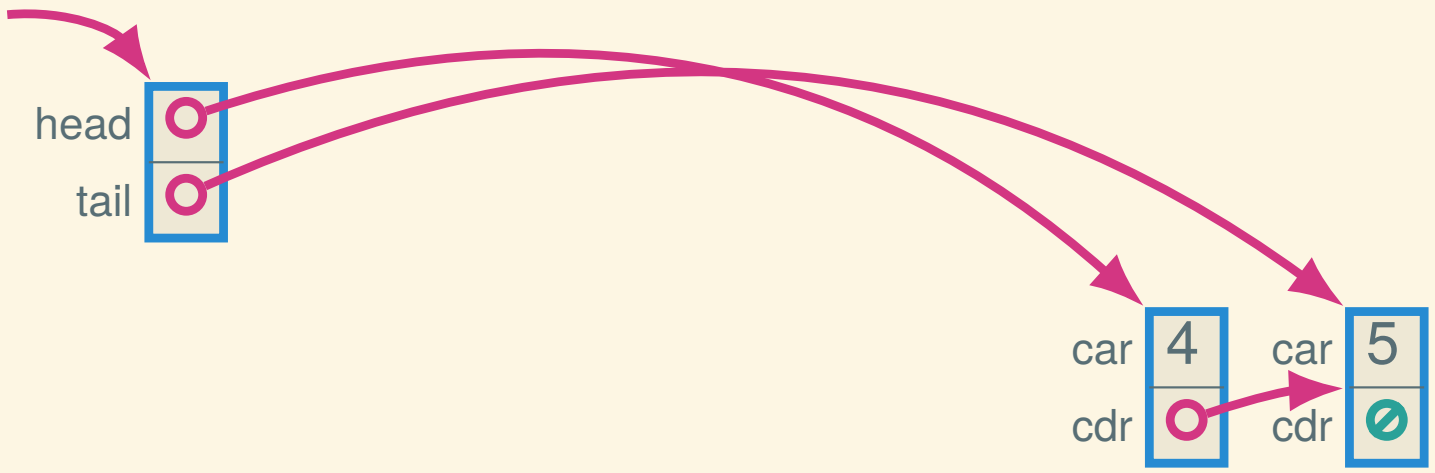

```
let q = LinkedListQueue()
q.enqueue(2)
q.enqueue(3)
q.enqueue(4)
q.enqueue(5)
```

Queue impl.: linked list with tail pointer



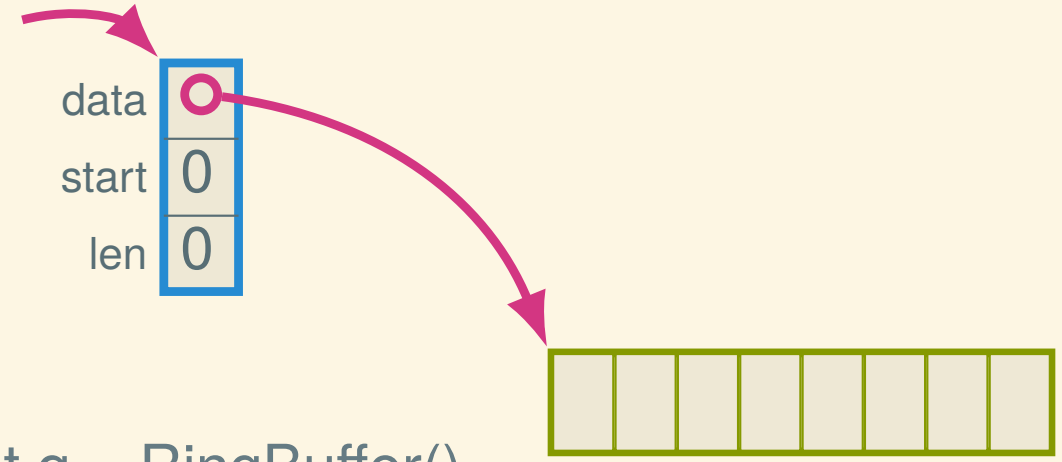

```
let q = LinkedListQueue()
q.enqueue(2)
q.enqueue(3)
q.enqueue(4)
q.enqueue(5)
q.dequeue()
```

Queue impl.: linked list with tail pointer



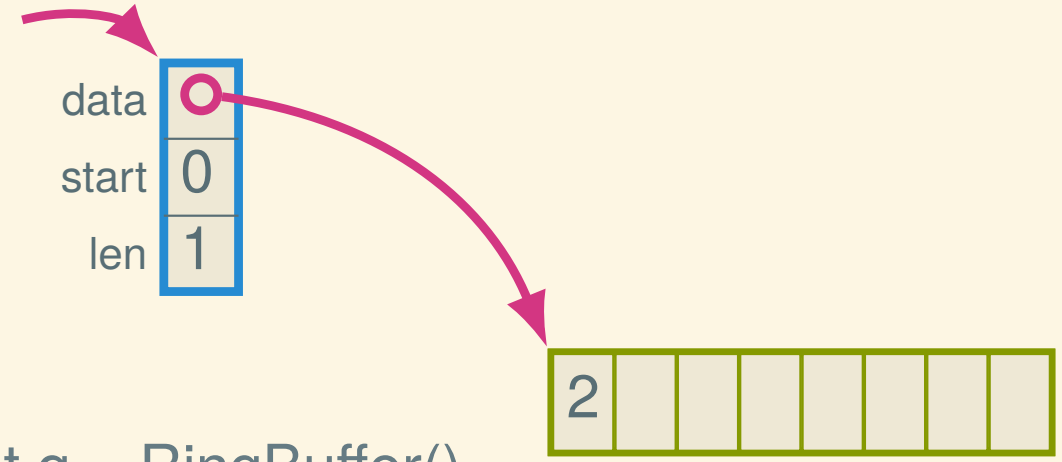

```
let q = LinkedListQueue()
q.enqueue(2)
q.enqueue(3)
q.enqueue(4)
q.enqueue(5)
q.dequeue()
q.dequeue()
```

Queue implementation: ring buffer



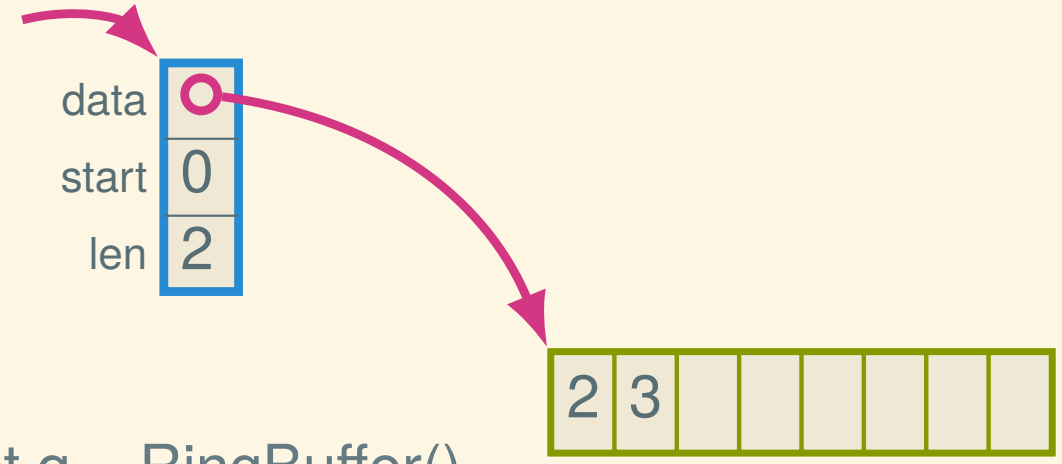
let q = RingBuffer()

Queue implementation: ring buffer



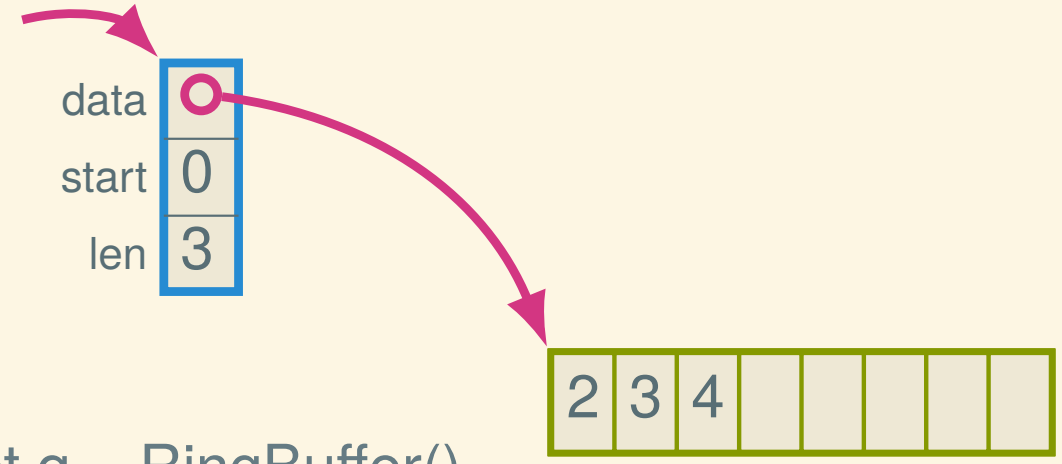

```
let q = RingBuffer()
q.enqueue(2)
```

Queue implementation: ring buffer



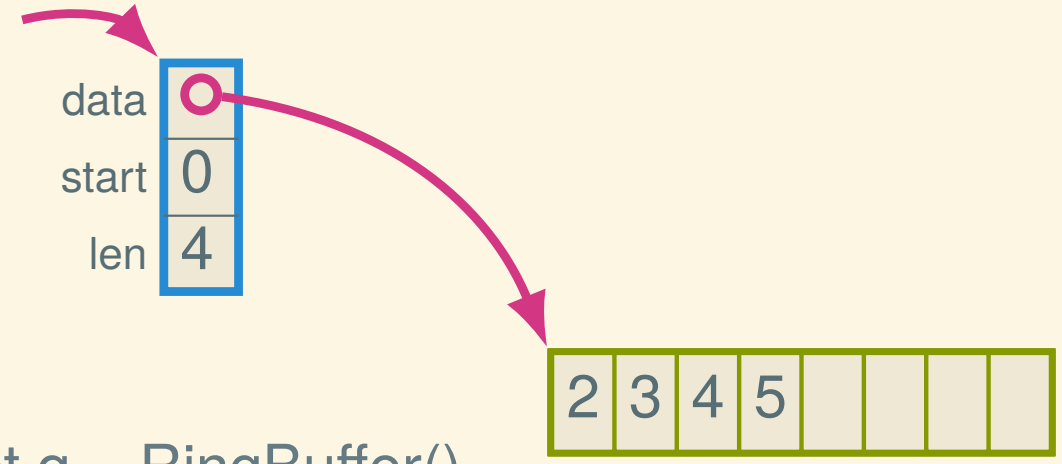

```
let q = RingBuffer()
q.enqueue(2)
q.enqueue(3)
```

Queue implementation: ring buffer



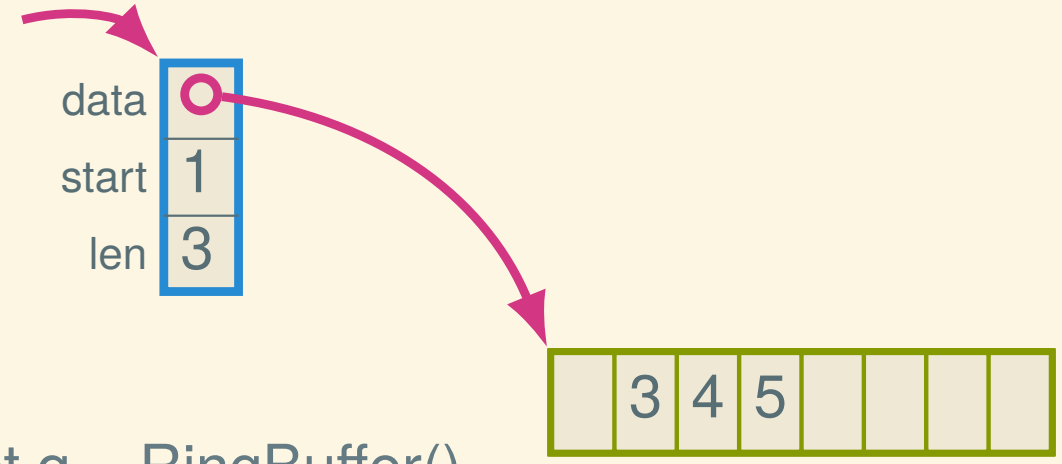

```
let q = RingBuffer()
q.enqueue(2)
q.enqueue(3)
q.enqueue(4)
```

Queue implementation: ring buffer



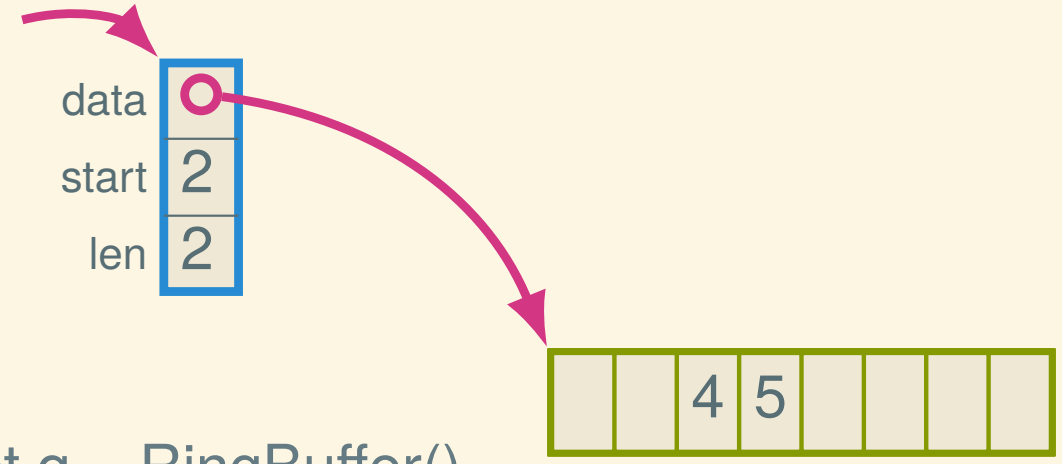

```
let q = RingBuffer()
q.enqueue(2)
q.enqueue(3)
q.enqueue(4)
q.enqueue(5)
```

Queue implementation: ring buffer



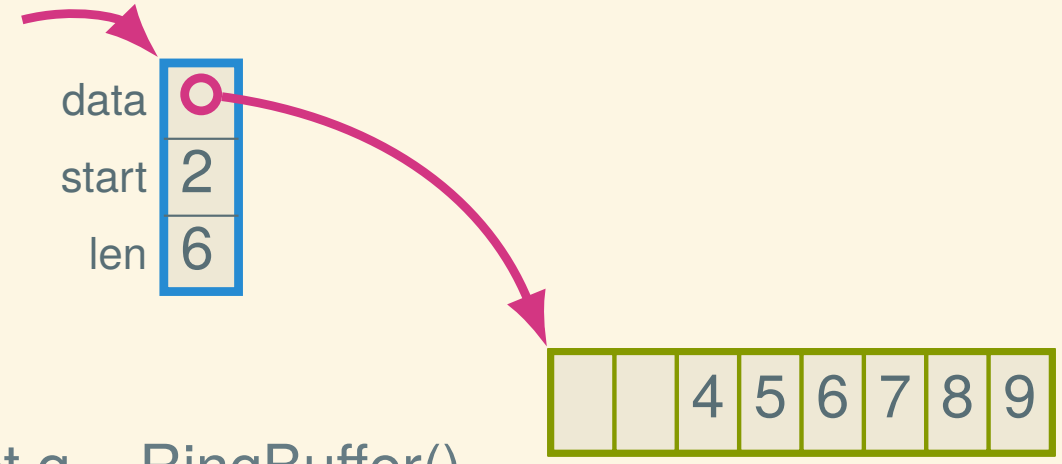

```
let q = RingBuffer()
q.enqueue(2)
q.enqueue(3)
q.enqueue(4)
q.enqueue(5)
q.dequeue()
```

Queue implementation: ring buffer



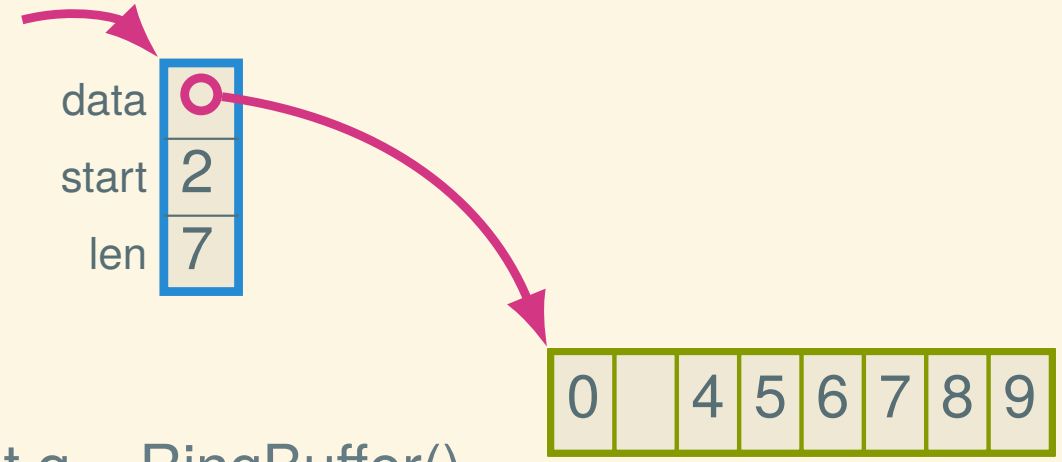

```
let q = RingBuffer()
q.enqueue(2)
q.enqueue(3)
q.enqueue(4)
q.enqueue(5)
q.dequeue()
q.dequeue()
```

Queue implementation: ring buffer



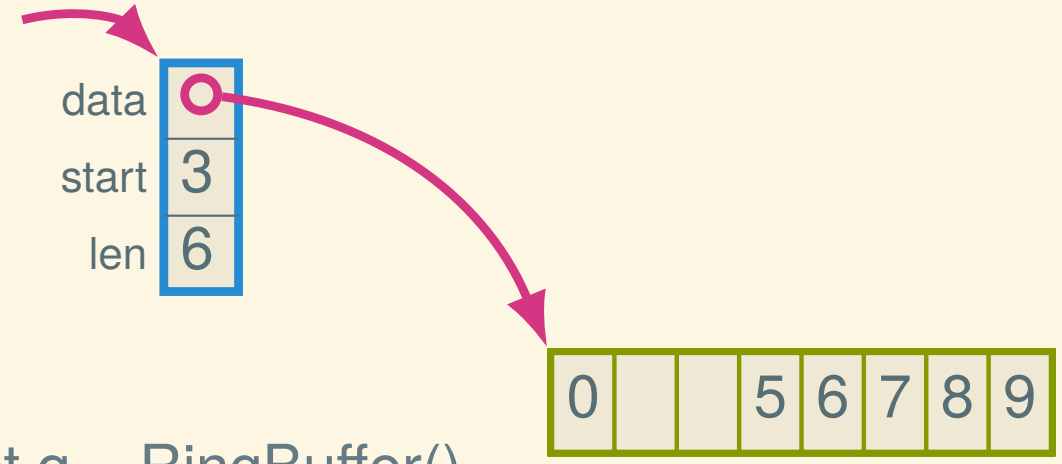

```
let q = RingBuffer()
q.enqueue(2)
q.enqueue(3)
q.enqueue(4)
q.enqueue(5)
q.dequeue()
q.dequeue()
⋮
```

Queue implementation: ring buffer




```
let q = RingBuffer()
q.enqueue(2)
q.enqueue(3)
q.enqueue(4)
q.enqueue(5)
q.dequeue()
q.dequeue()
⋮
q.enqueue(0)
```

Queue implementation: ring buffer




```
let q = RingBuffer()
q.enqueue(2)
q.enqueue(3)
q.enqueue(4)
q.enqueue(5)
q.dequeue()
q.dequeue()
⋮
q.enqueue(0)
q.dequeue()
```

Trade-offs: linked list queue versus ring buffer

Basically the same as for the stack implementations:

- Ring buffer has better constant factors and uses less space (potentially)
- Linked list doesn't fill up

Ring buffer in DSSL2

Signature, with `full?`

```
interface QUEUE[T]:
    def enqueue(self, element: T) -> VoidC
    def dequeue(self) -> T
    def empty?(self) -> bool?
    def full?(self) -> bool?
```

Representation and initialization

```
class RingBuffer (QUEUE):
    let data
    let start
    let size

    def __init__(self, capacity):
        self.data = [False; capacity]
        self.start = 0
        self.size = 0

    …
```

Size stuff

```
class RingBuffer (QUEUE):
    let data
    let start
    let size
    …

    def cap(self):
        self.data.len()

    def len(self):
        self.size

    def empty?(self):
        self.len() == 0

    def full?(self):
        self.len() == self.cap()

    …
```

Enqueueing

```
class RingBuffer (QUEUE):
    let data
    let start
    let size
    …

    def enqueue(self, element):
        if self.full?():
                error('RingBuffer.enqueue: full')
        let index = (self.start + self.size) % self.cap()
        self.data[index] = element
        self.size = self.size + 1

    …
```

Dequeueing

```
class RingBuffer (QUEUE):
    let data
    let start
    let size
    …

    def dequeue(self):
        if self.empty?():
            error('RingBuffer.dequeue: empty')
        let result = self.data[self.start]
        self.data[self.start] = False
        self.size = self.size - 1
        self.start = (self.start + 1) % self.cap()
        result

    …
```

Next time: BSTs and the Dictionary ADT