Abstract Data Types

EECS 214, Fall 2017

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: $|3\ 4\ 5\rangle$

ADT: Stack

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

Signature:

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

ADT: Queue (FIFO)

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

ADT: Queue (FIFO)

Looks like: ⟨3 4 5⟨

Signature:

- *enqueue*(Queue, Element): Void
- *dequeue*(Queue): Element
- *isEmpty*(Queue): Bool

Stack versus Queue

Stack signature:

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

Queue signature:

- *enqueue*(Queue, Element): Void
- *dequeue*(Queue): Element
- *isEmpty*(Queue): 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:

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

Laws:

$$\mathit{isEmpty}(|\rangle) \Rightarrow \top$$

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

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

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

ADT: Queue (FIFO)

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

Signature:

- *enqueue*(Queue, Element): Void
- *dequeue*(Queue): Element
- *isEmpty*(Queue): Bool

Laws:

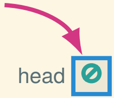
$$\mathit{isEmpty}(\langle\langle) \Rightarrow \top$$

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

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

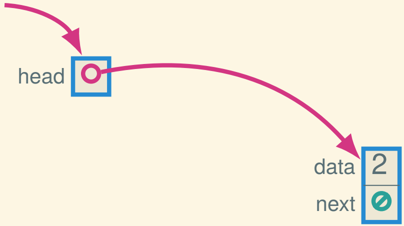
$$\{q = \langle e_1 e_2 \ldots e_k \langle\}\ \mathit{dequeue}(q) \Rightarrow e_1\ \{q = \langle e_2 \ldots e_k \langle\}$$

Stack implementation: linked list



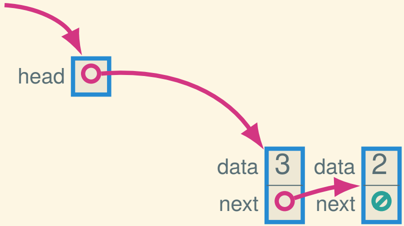
let s = new_stack()

Stack implementation: linked list



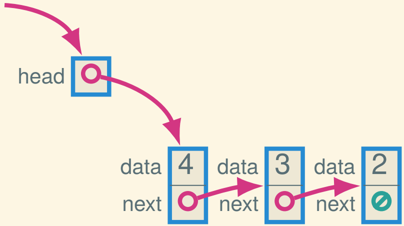
```
let s = new_stack()
push(s, 2)
```

Stack implementation: linked list



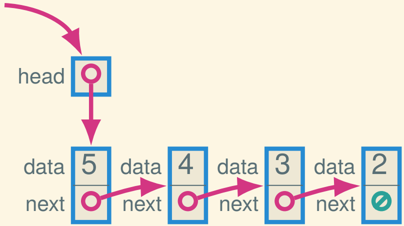
```
let s = new_stack()
push(s, 2)
push(s, 3)
```

Stack implementation: linked list



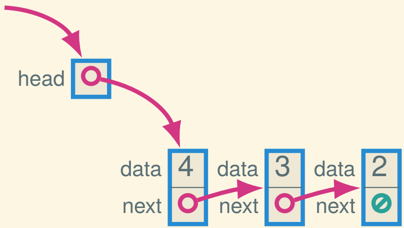
```
let s = new_stack()
push(s, 2)
push(s, 3)
push(s, 4)
```

Stack implementation: linked list



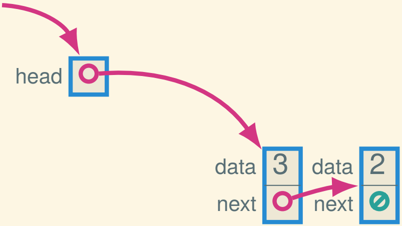
```
let s = new_stack()
push(s, 2)
push(s, 3)
push(s, 4)
push(s, 5)
```

Stack implementation: linked list



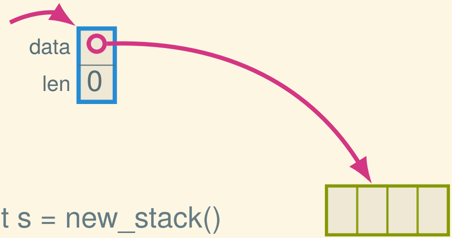
```
let s = new_stack()
push(s, 2)
push(s, 3)
push(s, 4)
push(s, 5)
pop(s)
```

Stack implementation: linked list



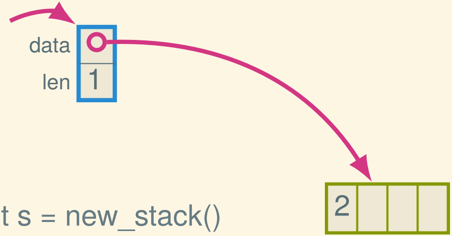
```
let s = new_stack()
push(s, 2)
push(s, 3)
push(s, 4)
push(s, 5)
pop(s)
pop(s)
```

Stack implementation: array



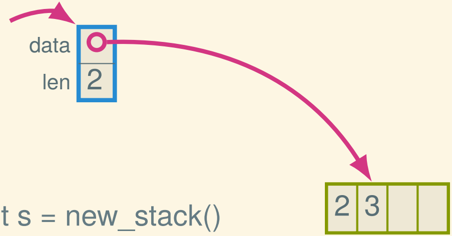
let s = new_stack()

Stack implementation: array



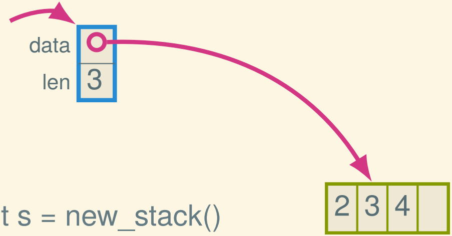
let s = new_stack()
push(s, 2)

Stack implementation: array



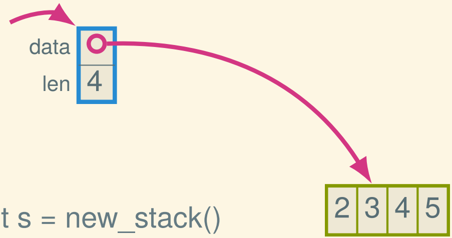
let s = new_stack()
push(s, 2)
push(s, 3)

Stack implementation: array



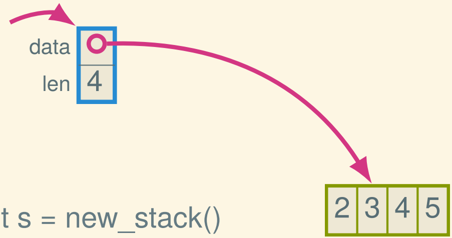
```
let s = new_stack()
push(s, 2)
push(s, 3)
push(s, 4)
```

Stack implementation: array



```
let s = new_stack()
push(s, 2)
push(s, 3)
push(s, 4)
push(s, 5)
```

Stack implementation: array



let s = new_stack()
push(s, 2)
push(s, 3)
push(s, 4)
push(s, 5)
push(s, 6)

ADT: Stack

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

Signature:

- *push*(Stack, Element): Void — $\mathcal{O}(1)$
- *pop*(Stack): Element — $\mathcal{O}(1)$
- *isEmpty*(Stack): Bool — $\mathcal{O}(1)$

Laws:

$$\mathit{isEmpty}(|\rangle) \Rightarrow \top$$

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

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

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

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:

- *enqueue*(Queue, Element): Void — $\mathcal{O}(1)$
- *dequeue*(Queue): Element — $\mathcal{O}(1)$
- *isEmpty*(Queue): Bool — $\mathcal{O}(1)$

Laws:

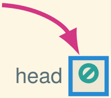
$$\mathit{isEmpty}(\langle\langle) \Rightarrow \top$$

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

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

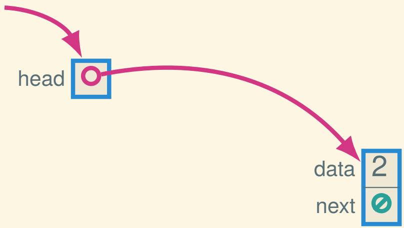
$$\{q = \langle e_1 e_2 \ldots e_k \langle\}\ \mathit{dequeue}(q) \Rightarrow e_1\ \{q = \langle e_2 \ldots e_k \langle\}$$

Queue implementation: linked list?



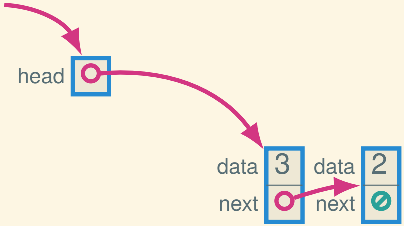
```
let q = new_queue()
```

Queue implementation: linked list?



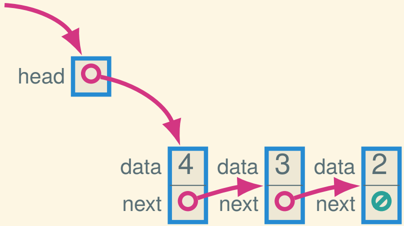
```
let q = new_queue()
enqueue(q, 2)
```

Queue implementation: linked list?



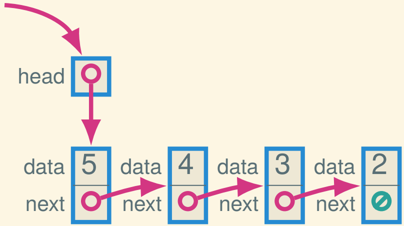
let q = new_queue()
enqueue(q, 2)
enqueue(q, 3)

Queue implementation: linked list?



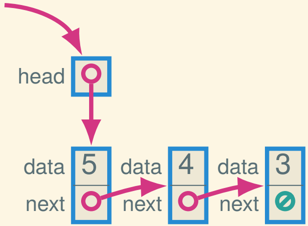
```
let q = new_queue()
enqueue(q, 2)
enqueue(q, 3)
enqueue(q, 4)
```

Queue implementation: linked list?



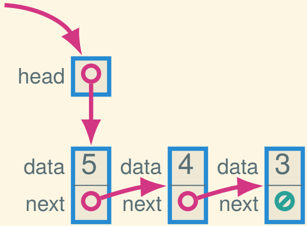
```
let q = new_queue()
enqueue(q, 2)
enqueue(q, 3)
enqueue(q, 4)
enqueue(q, 5)
```

Queue implementation: linked list?



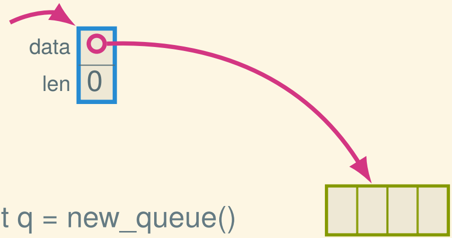
```
let q = new_queue()
enqueue(q, 2)
enqueue(q, 3)
enqueue(q, 4)
enqueue(q, 5)
dequeue(q)
```

Queue implementation: linked list?



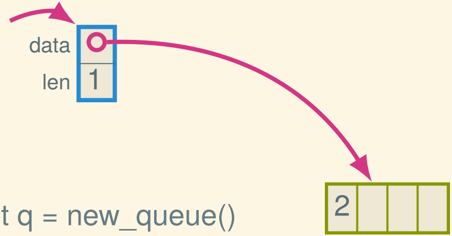
```
let q = new_queue()
enqueue(q, 2)
enqueue(q, 3)
enqueue(q, 4)
enqueue(q, 5)
dequeue(q) — 𝒪(n)?
```

Queue implementation: array?



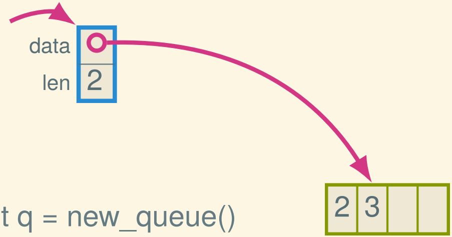
let q = new_queue()

Queue implementation: array?



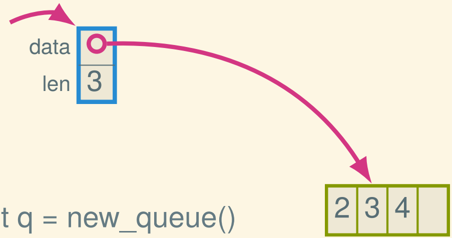
let q = new_queue()
enqueue(q, 2)

Queue implementation: array?



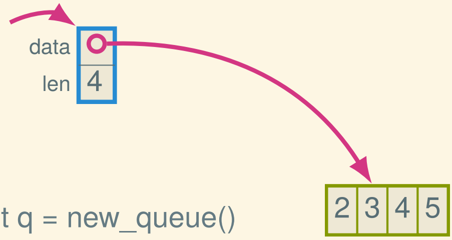
```
let q = new_queue()
enqueue(q, 2)
enqueue(q, 3)
```

Queue implementation: array?



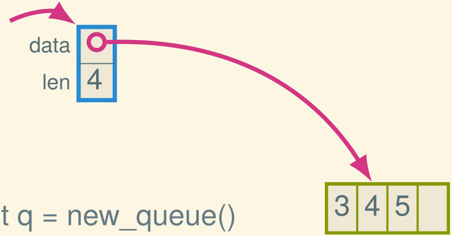
```
let q = new_queue()
enqueue(q, 2)
enqueue(q, 3)
enqueue(q, 4)
```

Queue implementation: array?



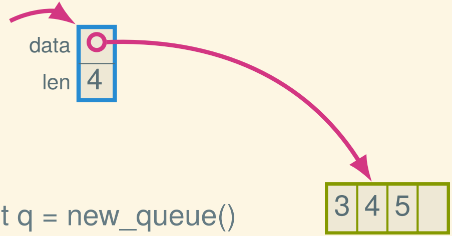
let q = new_queue()
enqueue(q, 2)
enqueue(q, 3)
enqueue(q, 4)
enqueue(q, 5)

Queue implementation: array?



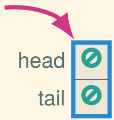
```
let q = new_queue()
enqueue(q, 2)
enqueue(q, 3)
enqueue(q, 4)
enqueue(q, 5)
s.dequeue()
```

Queue implementation: array?



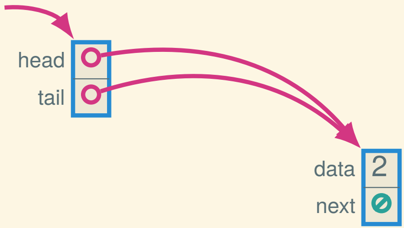
let q = new_queue()
enqueue(q, 2)
enqueue(q, 3)
enqueue(q, 4)
enqueue(q, 5)
s.dequeue() — $\mathcal{O}(n)$???

Queue impl.: linked list with tail pointer



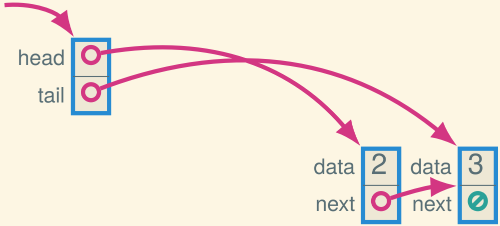
let q = new_queue()

Queue impl.: linked list with tail pointer



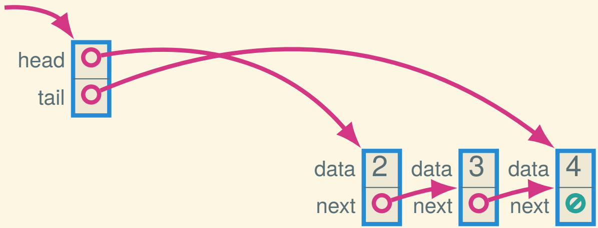
```
let q = new_queue()
enqueue(q, 2)
```

Queue impl.: linked list with tail pointer



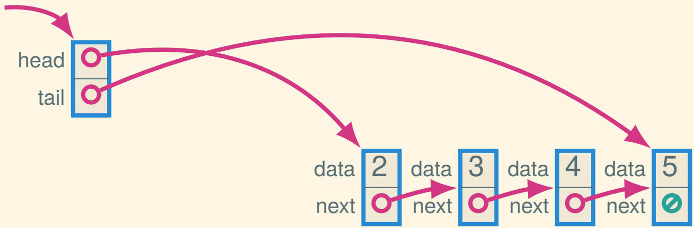
```
let q = new_queue()
enqueue(q, 2)
enqueue(q, 3)
```

Queue impl.: linked list with tail pointer



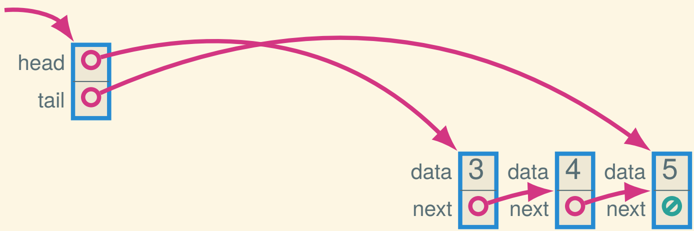
```
let q = new_queue()
enqueue(q, 2)
enqueue(q, 3)
enqueue(q, 4)
```

Queue impl.: linked list with tail pointer



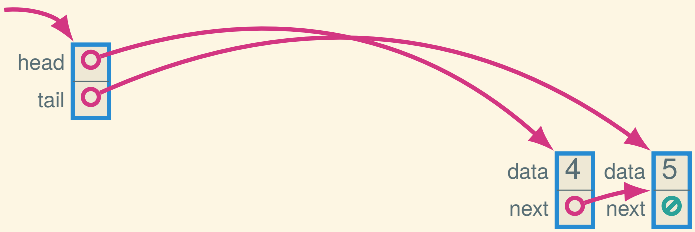
```
let q = new_queue()
enqueue(q, 2)
enqueue(q, 3)
enqueue(q, 4)
enqueue(q, 5)
```

Queue impl.: linked list with tail pointer



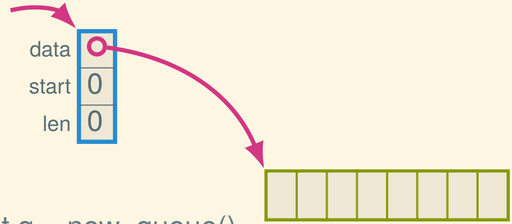
```
let q = new_queue()
enqueue(q, 2)
enqueue(q, 3)
enqueue(q, 4)
enqueue(q, 5)
dequeue(q)
```

Queue impl.: linked list with tail pointer



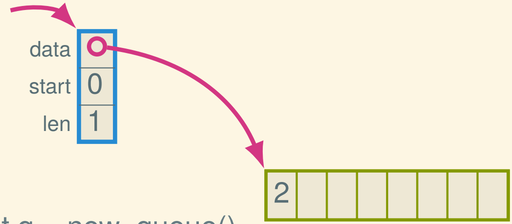
```
let q = new_queue()
enqueue(q, 2)
enqueue(q, 3)
enqueue(q, 4)
enqueue(q, 5)
dequeue(q)
dequeue(q)
```

Queue implementation: ring buffer



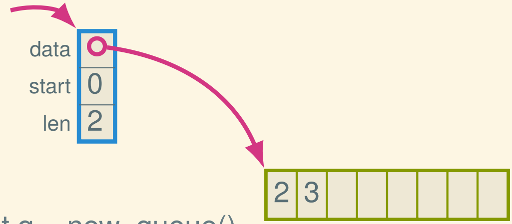
let q = new_queue()

Queue implementation: ring buffer



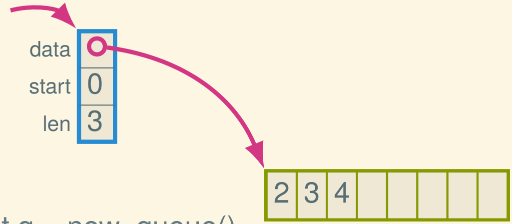
```
let q = new_queue()
enqueue(q, 2)
```

Queue implementation: ring buffer



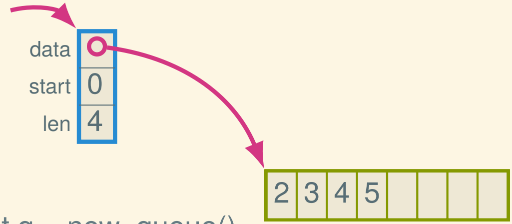
```
let q = new_queue()
enqueue(q, 2)
enqueue(q, 3)
```

Queue implementation: ring buffer



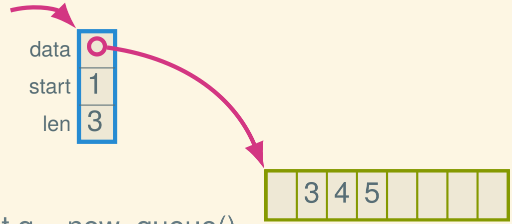
```
let q = new_queue()
enqueue(q, 2)
enqueue(q, 3)
enqueue(q, 4)
```

Queue implementation: ring buffer



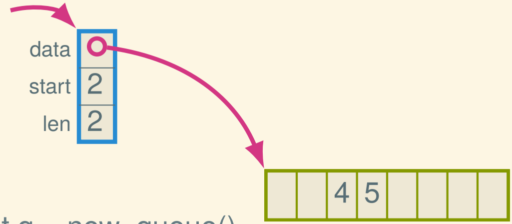
```
let q = new_queue()
enqueue(q, 2)
enqueue(q, 3)
enqueue(q, 4)
enqueue(q, 5)
```

Queue implementation: ring buffer



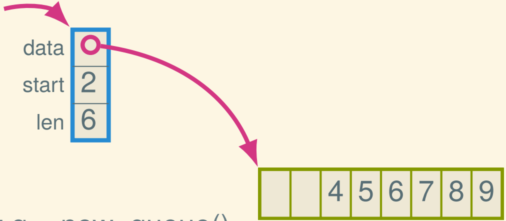
let q = new_queue()
enqueue(q, 2)
enqueue(q, 3)
enqueue(q, 4)
enqueue(q, 5)
dequeue(q)

Queue implementation: ring buffer



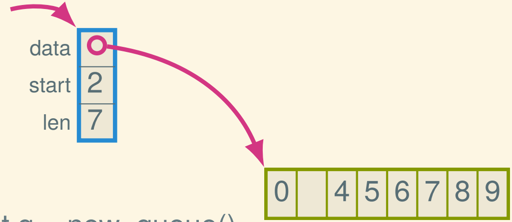
```
let q = new_queue()
enqueue(q, 2)
enqueue(q, 3)
enqueue(q, 4)
enqueue(q, 5)
dequeue(q)
dequeue(q)
```

Queue implementation: ring buffer



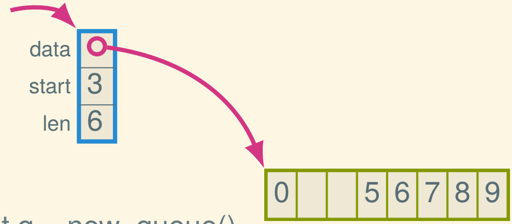
```
let q = new_queue()
enqueue(q, 2)
enqueue(q, 3)
enqueue(q, 4)
enqueue(q, 5)
dequeue(q)
dequeue(q)
⋮
```

Queue implementation: ring buffer



```
let q = new_queue()
enqueue(q, 2)
enqueue(q, 3)
enqueue(q, 4)
enqueue(q, 5)
dequeue(q)
dequeue(q)
⋮
enqueue(q, 0)
```

Queue implementation: ring buffer



```
let q = new_queue()
enqueue(q, 2)
enqueue(q, 3)
enqueue(q, 4)
enqueue(q, 5)
dequeue(q)
dequeue(q)
⋮
enqueue(q, 0)
dequeue(q)
```

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

Representation

```
# A QueueOf[X] is
#   queue(VectorOf[X or False], Natural, Natural)
# Interpretation:
# - `data` contains the elements of the queue,
# - `start` is the index of the first element, and
# - `size` is the number of elements.
defstruct queue(data, start, size)
```

Representation

```
# A QueueOf[X] is
#   queue(VectorOf[X or False], Natural, Natural)
# Interpretation:
# - `data` contains the elements of the queue,
# - `start` is the index of the first element, and
# - `size` is the number of elements.
defstruct queue(data, start, size)

let F = False
let QUEUE0 = queue([F; 8], 0, 0))
let QUEUE1 = queue([3,  F, F, F, F, F], 0, 1)
let QUEUE2 = queue([3,  4, F, F, F, F], 0, 2)
let QUEUE3 = queue([F,  F, 5, 6, 7, F], 2, 3)
let QUEUE4 = queue([9, 10, F, F, 7, 8], 4, 4)
```

Creating a new queue

```
# new_queue : Natural -> QueueOf[X]
def new_queue(capacity):
    queue([ False; capacity ], 0, 0)
```

Finding out the size and capacity

```
# queue_size : QueueOf[X] -> Natural
def queue_size(q): q.size

# queue_capacity : QueueOf[X] -> Natural
def queue_capacity(q): len(q.data)
```

Finding out the size and capacity

```
# queue_size : QueueOf[X] -> Natural
def queue_size(q): q.size

# queue_capacity : QueueOf[X] -> Natural
def queue_capacity(q): len(q.data)

# queue_empty? : QueueOf[X] -> Bool
def queue_empty?(q):
    queue_size(q) == 0

# queue_full? : QueueOf[X] -> Bool
def queue_full?(q):
    queue_size(q) == queue_capacity(q)
```

Enqueueing

```
# enqueue! : QueueOf[X] X -> Void
def enqueue!(q, element):
    if queue_full?(q):
        error('enqueue!: queue is full')
    let cap = queue_capacity(q)
    q.data[(q.size + q.start) % cap] = element
    q.size = q.size + 1
```

Dequeueing

```
# dequeue! : QueueOf[X] -> X
def dequeue!(q):
    if queue_empty?(q):
        error('dequeue!: queue is empty')
    let result = q.data[q.start]
    q.data[q.start] = False
    q.size = q.size - 1
    q.start = (q.start + 1) % queue_capacity(q)
    result
```

Next time: BSTs and the Dictionary ADT