Stacks and Queues

Lists with Rules for Removing Elements

Sometimes the order of our lists represents the order our data

- was added to the list
- must be removed from the list

Stacks follow the **First In Last Out** rule

Queues are **First In First Out**

Stacks: Push and Pop

Standard computer science terminology:

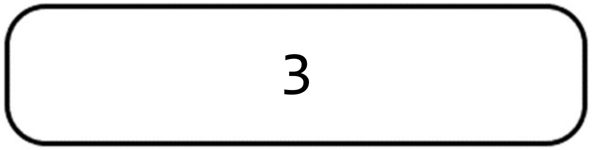
- **push** to add something to the ‘top’ of a stack
- **pop** to remove something from the top of a stack

Stacks: Push and Pop

Standard computer science terminology:

- **push** to add something to the ‘top’ of a stack
- **pop** to remove something from the top of a stack

```
push  3
```



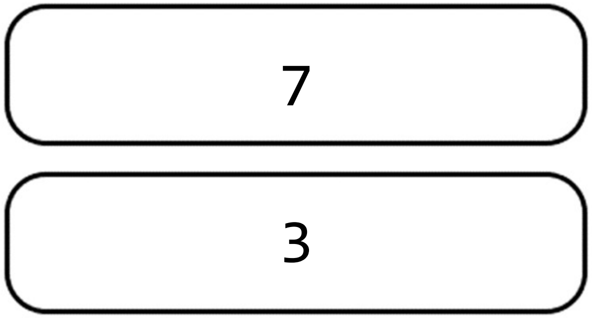
3

Stacks: Push and Pop

Standard computer science terminology:

- **push** to add something to the ‘top’ of a stack
- **pop** to remove something from the top of a stack

```
push 7
```



7

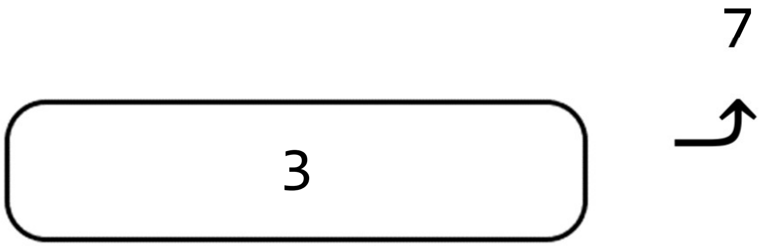
3

Stacks: Push and Pop

Standard computer science terminology:

- **push** to add something to the 'top' of a stack
- **pop** to remove something from the top of a stack

pop

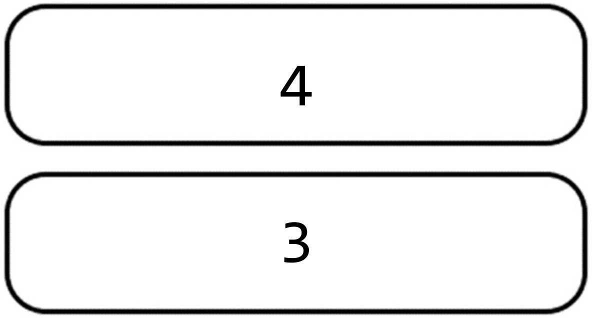


Stacks: Push and Pop

Standard computer science terminology:

- **push** to add something to the ‘top’ of a stack
- **pop** to remove something from the top of a stack

```
push  4
```



4

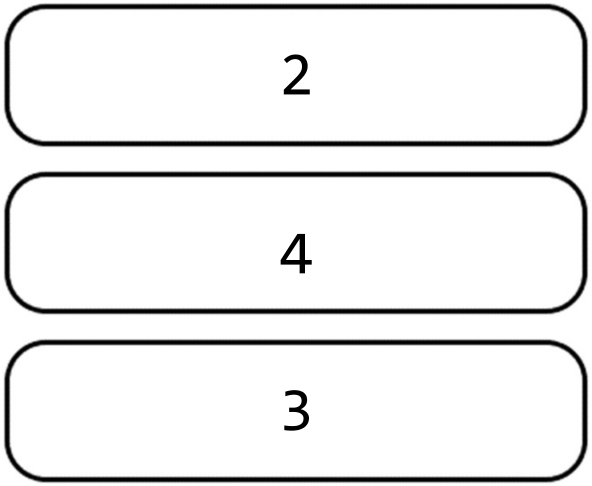
3

Stacks: Push and Pop

Standard computer science terminology:

- **push** to add something to the 'top' of a stack
- **pop** to remove something from the top of a stack

```
push  2
```



2

4

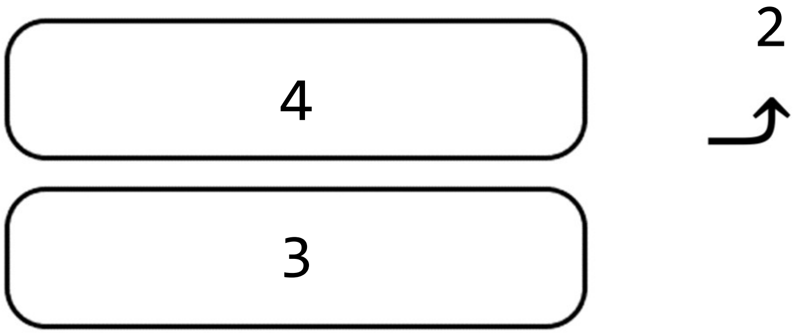
3

Stacks: Push and Pop

Standard computer science terminology:

- **push** to add something to the 'top' of a stack
- **pop** to remove something from the top of a stack

pop



Stacks in Haskell

We would like to implement stacks so that both push and pop are fast.

The List structure we already know does this excellently

```
push :: a -> [a] -> [a]
push x xs = x : xs

pop :: [a] -> (a,[a])
pop stack = case stack of
  []   -> error "pop on empty stack"
  x:xs -> (x,xs)
```

Stacks in Haskell

We would like to implement stacks so that push and pop is fast.

The List structure we already know does this excellently

```
push :: a -> [a] -> [a]
push = (:)

pop :: [a] -> (a,[a])
pop stack = case stack of
  []   -> error "pop on empty stack"
  x:xs -> (x,xs)
```

Application of Stacks: Balanced Parentheses

How can we check expressions have balanced parentheses or not?

```
((5 + (3) * (4 + 6)))
((5 + (3) * (4 + 6))
((5 + (3) * (4 + 6))))
(5 + (3))) * ((4 + 6)
```

More than a matter of counting left and right (see last example)

Queues: Enqueue and Dequeue

Standard computer science terminology:

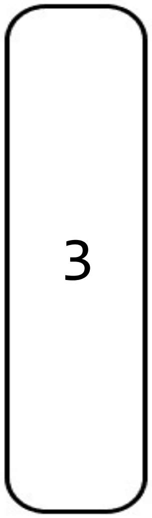
- **enqueue** to add something to the ‘back’ of a queue
- **dequeue** to remove something from the ‘front’ of a queue

Queues: Enqueue and Dequeue

Standard computer science terminology:

- **enqueue** to add something to the 'back' of a queue
- **dequeue** to remove something from the 'front' of a queue

enqueue 3

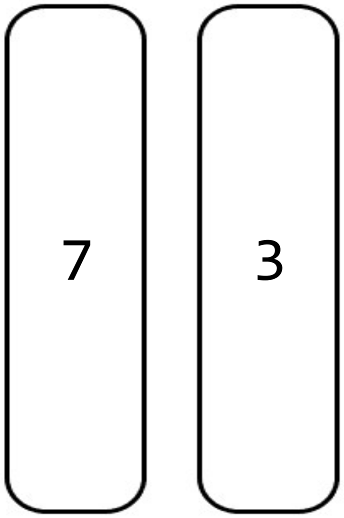


Queues: Enqueue and Dequeue

Standard computer science terminology:

- **enqueue** to add something to the 'back' of a queue
- **dequeue** to remove something from the 'front' of a queue

enqueue 7

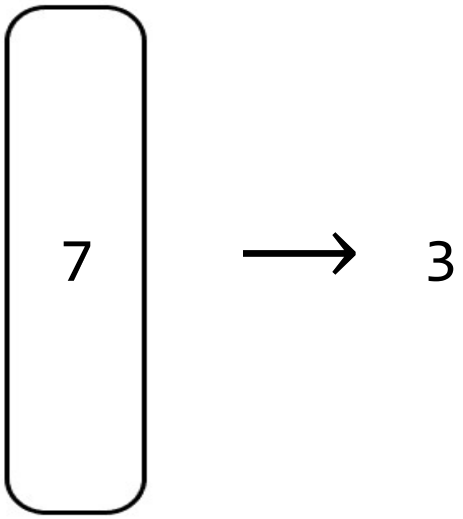


Queues: Enqueue and Dequeue

Standard computer science terminology:

- **enqueue** to add something to the 'back' of a queue
- **dequeue** to remove something from the 'front' of a queue

dequeue

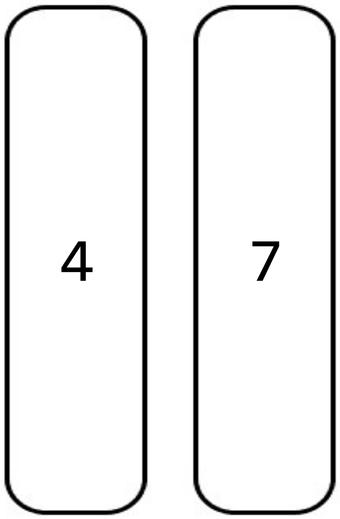


Queues: Enqueue and Dequeue

Standard computer science terminology:

- **enqueue** to add something to the 'back' of a queue
- **dequeue** to remove something from the 'front' of a queue

enqueue 4

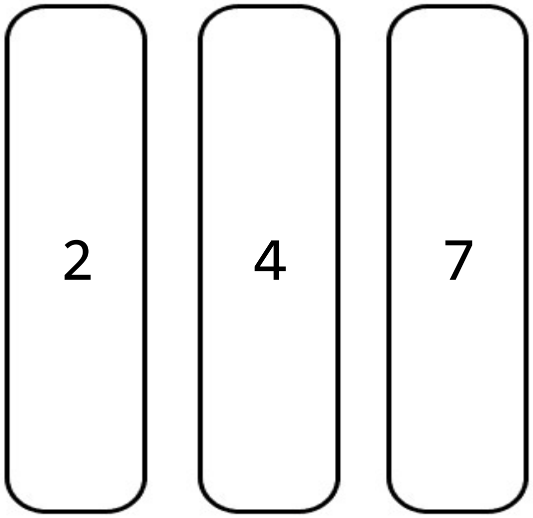


Queues: Enqueue and Dequeue

Standard computer science terminology:

- **enqueue** to add something to the 'back' of a queue
- **dequeue** to remove something from the 'front' of a queue

enqueue 2

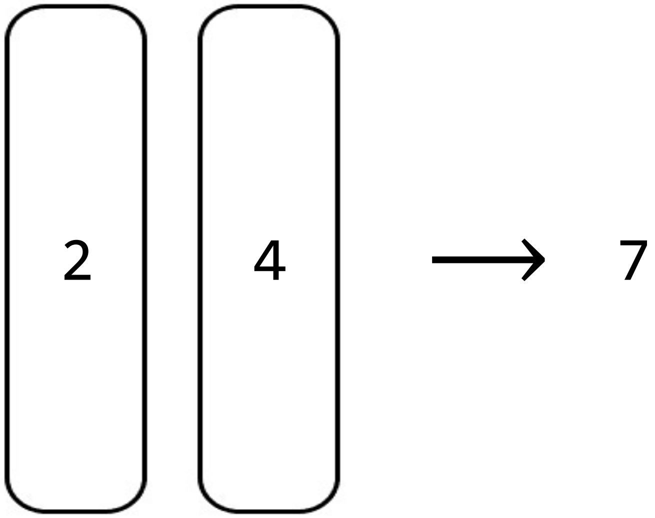


Queues: Enqueue and Dequeue

Standard computer science terminology:

- **enqueue** to add something to the 'back' of a queue
- **dequeue** to remove something from the 'front' of a queue

dequeue



Queues in Haskell

As we will see, it is not quite so obvious what is the *best* way to implement queues in Haskell

So we will specify the operations we expect as a **typeclass**, and compare two different instantiations.

```
class Queue q where
  enqueue  :: a -> q a -> q a
  dequeue  :: q a -> (a,q a)
  emptyQ   :: q a
  isEmptyQ :: q a -> Bool
```

Queues via Lists

We can implement queues, as with stacks, via lists:

```
instance Queue [] where
  …
```

Problem: we can implement *one* of enqueue or dequeue the same way we pushed and popped, but not both!

- If we enqueue to the left of the list, our dequeue might be slow…
- but if we dequeue from the left of our list, our enqueue might be slow!

Queues via Pairs of Lists

```
data TwoListQueue a = Queue [a] [a]

instance Queue TwoListQueue where
  …
```

We use the left hand list for output, and right hand for input

- Shuffling values from the output list to the input list only when we need to

Example

Starting with the completely empty Queue `[] []`

enqueue 3	…	`Queue [] [3]`
enqueue 7	…	`Queue [] [7,3]`
dequeue	…	`Queue [3,7] [] → Queue [7] []`
enqueue 4	…	`Queue [7] [4]`
enqueue 2	…	`Queue [7] [2,4]`
dequeue	…	`Queue [] [2,4]`

A Comparison of Different Implementations

Let’s define a function

```
stackToQueue :: Queue q => [a] -> q a
```

popping elements one by one off a stack (list) and enqueuing them into a queue as we go.

Then compare

```
> stackToQueue [1..20000] :: [Int]
> stackToQueue [1..20000] :: TwoListQueue Int
```

Using Queues

Suppose we want to write a functions involving the `Queue` typeclass

- e.g. stackToQueue :: `Queue q => Stack a -> q a` empties a stack into a queue

We cannot assume anything about how `Queue` has been implemented.

- Only use enqueue, `dequeue`, `emptyQ`, `isEmptyQ`
- If we try to refer to an empty queue as `[]`, or as `Queue [] []`, our code will not compile

This is a general feature of working with typeclasses: only use the functions we know must exist.