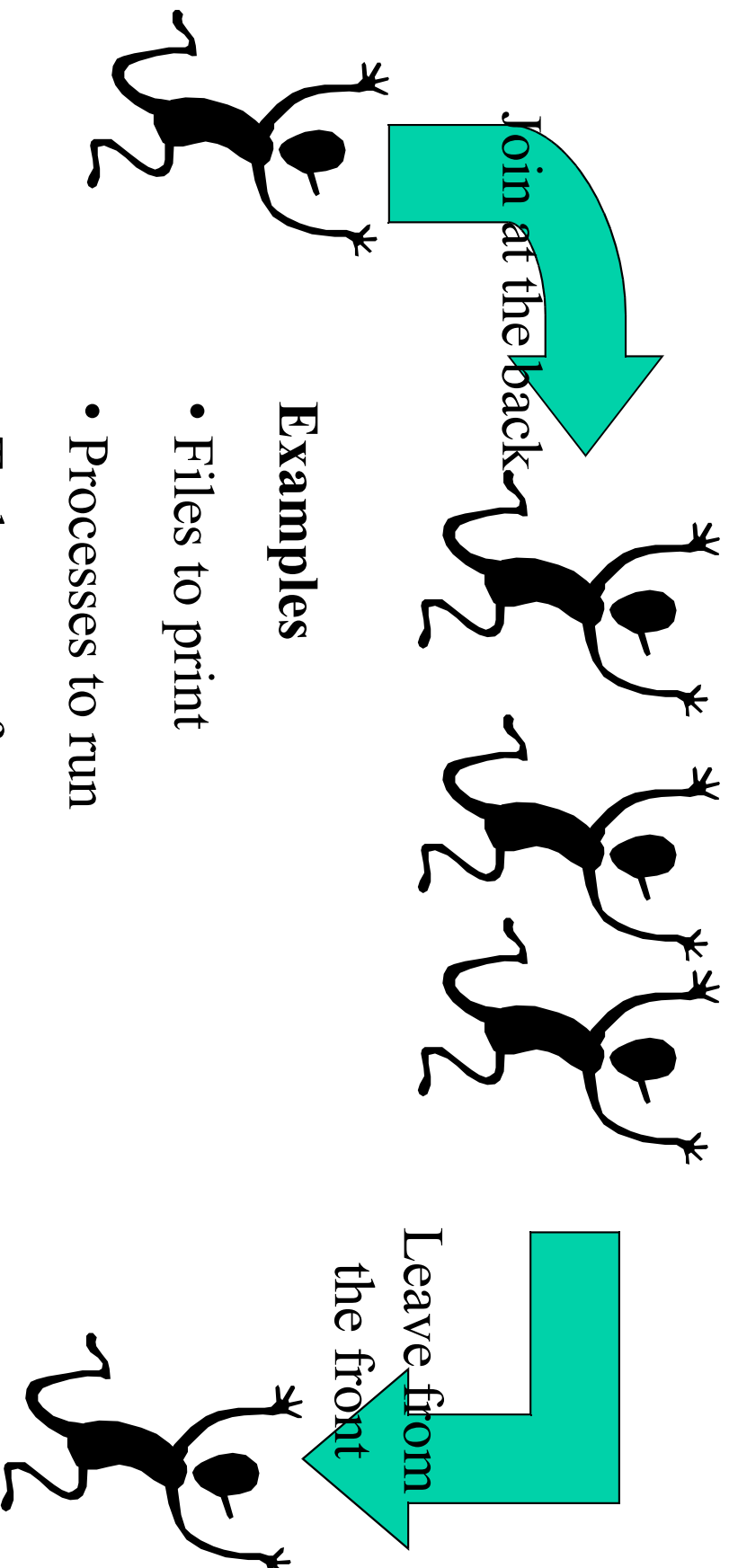


What is a Queue?



Examples

- Files to print
- Processes to run
- Tasks to perform

What is a Queue?

A *queue* contains a sequence of values. We can add elements at the back, and remove elements from the front.

We'll implement the following operations:

```
empty  :: Q a          -- an empty queue
add    :: a -> Q a -- add an element at the back
remove :: Q a -> Q a  -- remove an element from the front
front  :: Q a -> a    -- inspect the front element
isEmpty :: Q a -> Bool -- check if the queue is empty
```

First Try

data $Q\ a = Q\ [a]$ **deriving** (Eq, Show)

empty $= Q\ []$

add $x\ (Q\ xs)$ $= Q\ (xs++[x])$

remove $(Q\ (x:xs)) = Q\ xs$

front $(Q\ (x:xs))$ $= x$

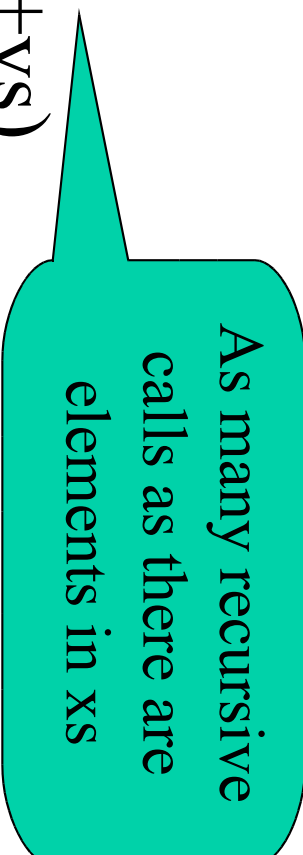
isEmpty $(Q\ xs)$ $= \text{null}\ xs$

Works, but slow

```
add x (Q xs) = Q (xs++[x])
```

```
[]    ++ ys = ys
```

```
(x:xs) ++ ys = x : (xs++ys)
```



As many recursive
calls as there are
elements in xs

Add 1, add 2, add 3, add 4, add 5...

Time is the *square* of the number of additions

A Module

- Implement the result in a *module*
- Use as specification
- Allows the re-use
 - By other programmers
 - Of the same names

SpecQueue Module

module SpecQueue **where**

data Q a = Q [a] **deriving** (Eq, Show)

empty = Q []

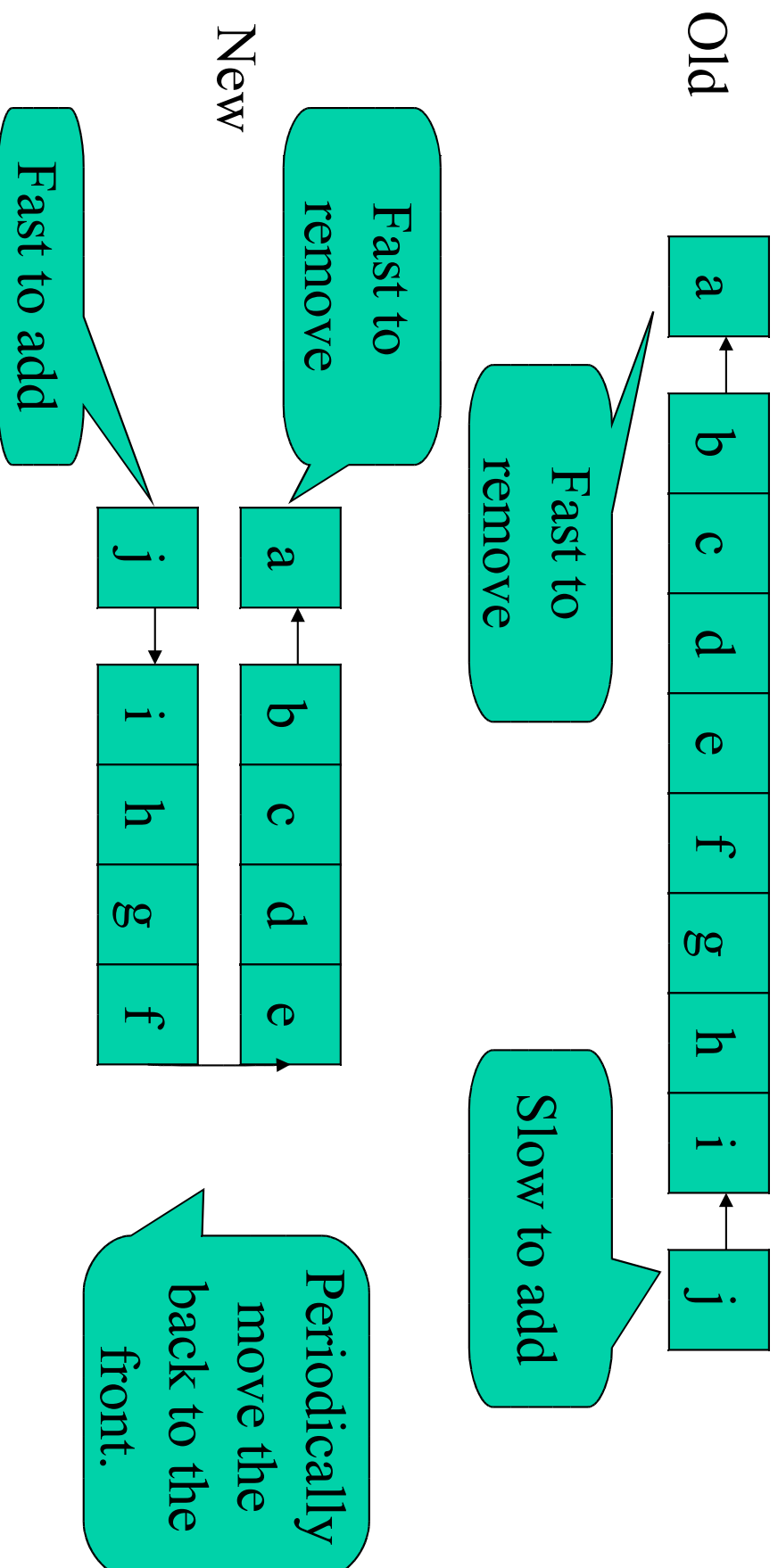
add x (Q xs) = Q (xs++[x])

remove (Q (x:xs)) = Q xs

front (Q (x:xs)) = x

isEmpty (Q xs) = null xs

New Idea: Store the Front and Back Separately



Fast Datatype

data $Q\ a = Q\ [a]\ [a]$

deriving (Eq, Show)



The front and the back
part of the queue.

Fast Operations

empty

||
O
[]
[]

add x (Q front back)

$$= \mathbf{Q}_{\text{front}}(\mathbf{x}:\text{back})$$

```
remove(Q(x:front) back) = fixQ front back
```

$$\text{front}(\text{Q}(\text{x:front}) \text{ back}) = \text{x}$$

```
isEmpty (Q front back) = null front && null back
```

Flip the queue when
we serve the last
person in the front

Smart Constructor

```
fixQ []    back = Q (reverse back) []  
fixQ front back = Q front back
```

This takes *one function call per element* in the back—each element is inserted into the back (one call), flipped (one call), and removed from the front (one call)

How can we test the fast functions?

- By using the original implementation as a *reference*
- The behaviour should be “the same”
 - Check results
- First version is an *abstract model* that is “obviously correct”

Comparing the Implementations

- They operate on different *types* of queues
- To compare, must convert between them
 - Can we convert a slow Q to a Q ?
 - Where should we split the front from the back???

contents (Q front back) = Q (front++reverse back)

- Retrieve the simple “model” contents from the implementation

Accessing modules

import qualified SpecQueue as Spec

contents :: Q Int -> Slow.Q Int

contents (Q front back) =

Spec.Q (front ++ reverse back)

The Properties

`prop_Empty =`

`contents empty == Slow.empty`

`prop_Add x q =`

`contents (add x q) == Slow.add x (contents q)`

`prop_Remove q =`

`contents (remove q) == Slow.remove (contents q)`

`prop_Front q =`

`front q == Slow.front (contents q)`

`prop_IsEmpty q =`

`isEmpty q == Slow.isEmpty (contents q)`

The behaviour is
the same, except
for type
conversion

Generating Qs

```
instance Arbitrary a => Arbitrary (Q a) where  
  arbitrary = do front <- arbitrary  
               back <- arbitrary  
               return (Q front back)
```

A Bug!

```
Queues> quickCheck prop_Remove
```

```
*** Failed! Exception: 'Queue.hs:22:0-42: Non-exhaustive patterns in  
function remove' (after 1 test):
```

```
Q [] []
```


Preconditions

- A condition that *must hold* before a function is called

prop_remove q =

not (isEmpty q) ==>

retrieve (remove q) == remove (retrieve q)

prop_front q =

not (isEmpty q) ==>

front q == front (retrieve q)

- Useful to be precise about these

Another Bug!

Queues> quickCheck prop_Remove

*** Failed! Exception: 'Queue.hs:22:0-42:

Non-exhaustive patterns in function remove'
(after 2 tests):

Q [] [-1,0]



But this ought not to happen!

An Invariant

- Q values ought *never* to have an empty front, and a non-empty back!
- Formulate an *invariant*
invariant (Q front back) =
???

An Invariant

- Q values ought *never* to have an empty front, and a non-empty back!
- Formulate an *invariant*
invariant (Q front back) =
not (null front && not (null back))

Testing the Invariant

`prop_Invariant :: Q Int -> Bool`

`prop_Invariant q = invariant q`

- Of course, it fails...

`Queues> quickCheck prop_invariant`

Falsifiable, after 4 tests:

`Q [] [-1]`

Fixing the Generator

```
instance Arbitrary a => Arbitrary (Q a) where  
  arbitrary = do front <- arbitrary  
               back <- arbitrary  
               return (Q front  
                     (if null front then [] else back))
```

- **Now** prop_Invariant passes the tests

Testing the Invariant

- We've *written down* the invariant
- We've seen to it that we only generate valid Qs as *test data*
- We must ensure that the *queue functions* only build valid Q values!
 - It is at this stage that the invariant is most useful

Invariant Properties

`prop_Empty_Inv =`

`invariant empty`

`prop_Add_Inv x q =`

`invariant (add x q)`

`prop_Remove_Inv q =`

`not (isEmpty q) ==>`

`invariant (remove q)`

A Bug in the Q operations!

Queues> quickCheck prop_Add_Inv

Falsifiable, after 2 tests:

0

Q [] []

Queues> add 0 (Q [] [])

Q [] [0]



The invariant is False!

Fixing add

add x (Q front back) = **fix**Q front (x:back)

- We must flip the queue when *the first element* is *inserted* into an empty queue
- Previous bugs were in our understanding (our properties)—this one is in our implementation code

Summary

- Data structures *store data*
- Obeying an *invariant*
- ... that functions and operations
 - can make use of (to search faster)
 - have to respect (to not break the invariant)
- Writing down and testing invariants and properties is a good way of finding errors