

Datastructures

Data Structures

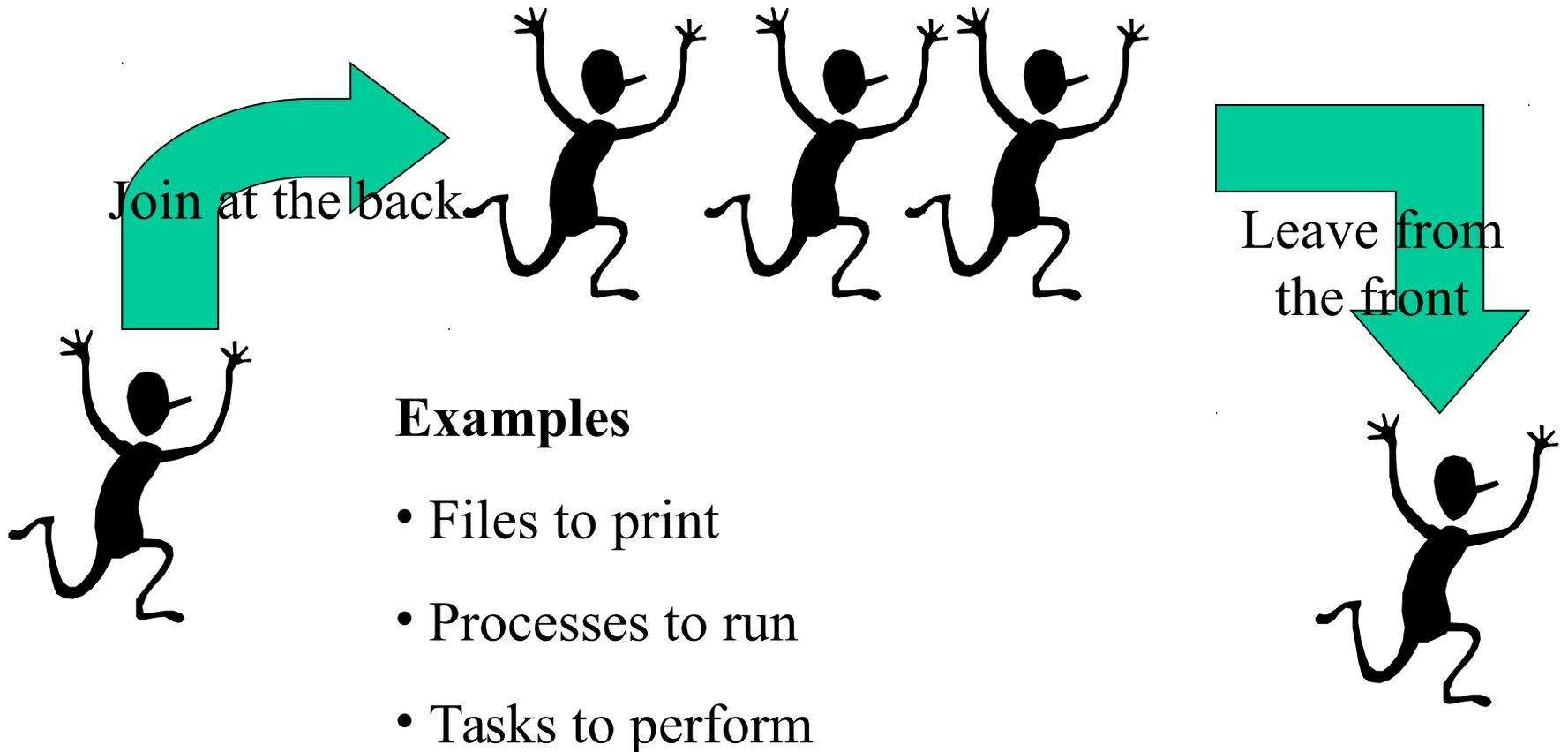
- Datatype
 - A model of something that we want to represent in our program
- Data structure
 - A particular way of *storing* data
 - How? Depending on what we want to do with the data
- Today: Two examples
 - Queues
 - Tables

Using QuickCheck to Develop Fast Queue Operations

What we're going to do:

- Explain what a *queue* is, and give *slow* implementations of the queue operations, to act as a specification.
- Give a fast implementation of the queue.
- Formulate properties that say the fast implementation is "correct".
- Test them with QuickCheck.

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 -> 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
```

new
type

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) = null xs

“Obviously”
correct

Works, but slow

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

$[] \quad ++ \text{ } ys = ys$

$(x:xs) ++ \text{ } ys = x : (xs++ys)$

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

Abstract data types

- Useful to separate the queue *interface* from the *implementation*
- Interface:

```
empty  :: Q a
add    :: a -> Q a -> Q a
remove :: Q a -> Q a
front  :: Q a -> a
isEmpty :: Q a -> Bool
```
- Implementation:

```
data Q a = ...
empty = ...
```
- Put the implementation in a *module*
- Allows programmers to switch implementation simply by changing imports

SlowQueue Module

module SlowQueue **where**

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

empty = Q []

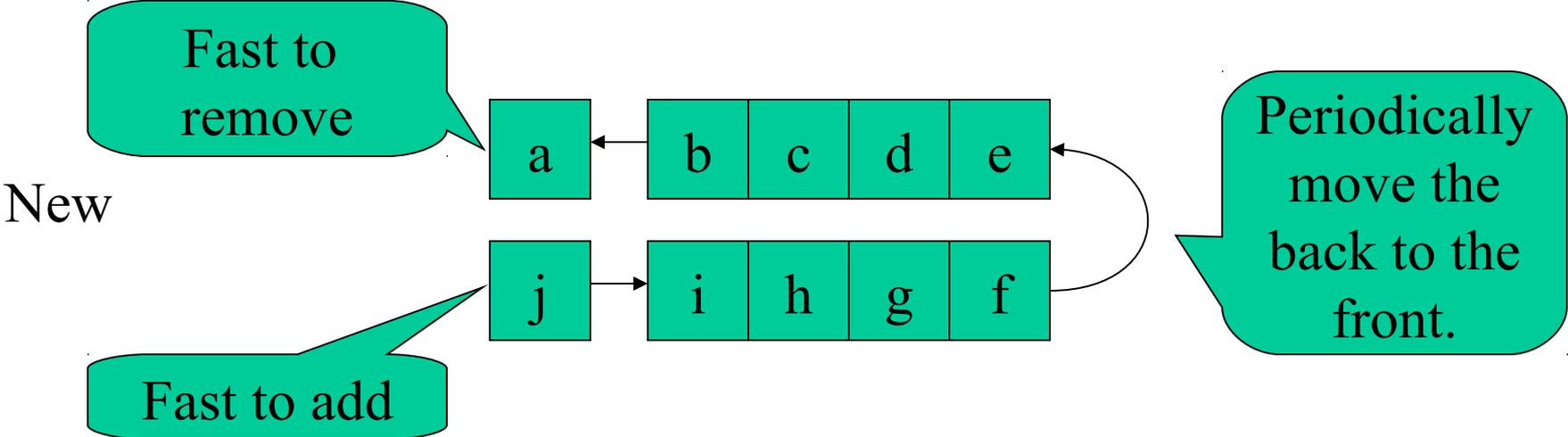
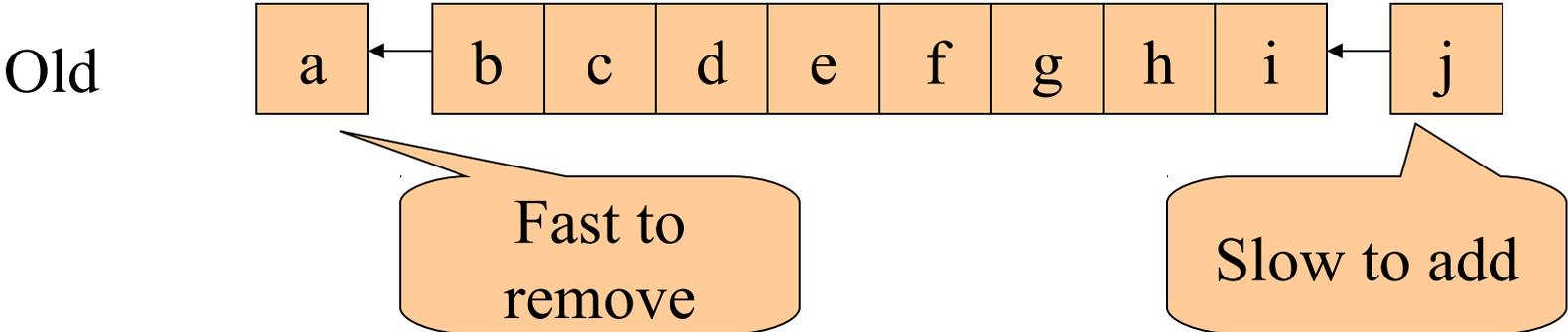
add x (Q xs) = Q (xs++[x])

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

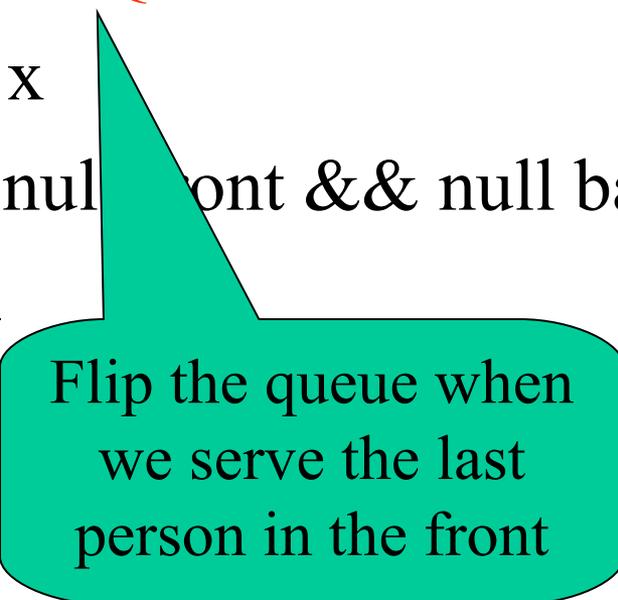
data Q a = Q [a] [a]
deriving (Eq, Show)



The front and the back
part of the queue.

Fast Operations

empty = Q [] []
add x (Q front back) = Q front (x:back)
remove (Q (x:front) back) = **fixQ** front back
front (Q (x:front) back) = 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???
 - Can we convert a Q to a slow Q?

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

- Retrieve the simple "model" contents from the implementation

Accessing modules

```
import qualified SlowQueue as Slow
```

```
contents :: Q Int -> Slow.Q Int
```

```
contents (Q front back) =
```

```
    Slow.Q (front ++ reverse back)
```



Qualified name

The Properties

The behaviour is
the same, except
for type
conversion

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)

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) ==>

contents (remove q) == remove (contents q)

prop_front q =

not (isEmpty q) ==>

front q == front (contents 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) =
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 made sure 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) = **fixQ** 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

Another Datastructure: Tables

A *table* holds a collection of *keys* and associated *values*.

For example, a phone book is a table whose keys are names, and whose values are telephone numbers.

Problem: Given a table and a key, find the associated value.

John Hughes	1001
Mary Sheeran	1013
Koen Claessen	5424
Hans Svensson	1079

Table Lookup Using Lists

Since a table may contain any kind of keys and values, define a parameterised type:

E.g. `[("x",1), ("y",2)] :: Table String Int`

type `Table k v = [(k, v)]`

`lookup "y" ...`
→ `Just 2`

`lookup :: Eq k => k -> Table k v -> Maybe v`

`lookup "z" ...`
→ `Nothing`

Finding Keys Fast

Finding keys by searching from the beginning is slow!

A better method:

look somewhere in the middle, and then look backwards or forwards depending on what you find.

(This assumes the table is sorted).

Claessen?

Aaboen A	
Nilsson Hans	
Östvall Eva	

Representing Tables

We must be able to break up a table fast, into:

- A smaller table of entries before the middle one,
- the middle entry,
- a table of entries after it.

data Table k v =

Join (Table k v) k v (Table k v)

Aaboen A	

Nilsson Hans	
--------------	--

Östvall Eva	

Quiz

What's wrong with this (recursive) type?

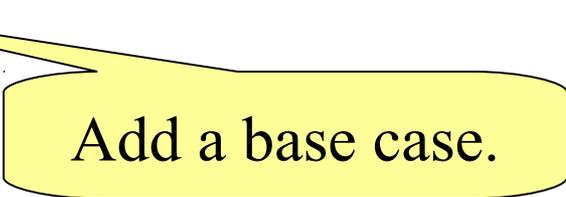
data Table k v = Join (Table k v) k v (Table k v)

Quiz

What's wrong with this (recursive) type? No base case!

```
data Table k v = Join (Table k v) k v (Table k v)
```

```
| Empty
```



Add a base case.

Looking Up a Key

To look up a key in a table:

- If the table is empty, then the key is not found.
- Compare the key with the key of the middle element.
- If they are equal, return the associated value.
- If the key is less than the key in the middle, look in the first half of the table.
- If the key is greater than the key in the middle, look in the second half of the table.

Quiz

Define

`lookupT :: Ord k => k -> Table k v -> Maybe v`

Recall

data Table k v = Join (Table k v) k v (Table k v)
| Empty

Quiz

Define

`lookupT :: Ord k => k -> Table k v -> Maybe v`

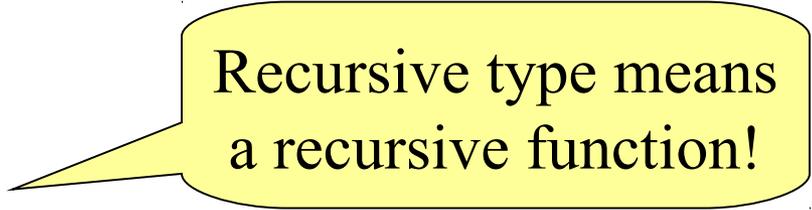
`lookupT key Empty = Nothing`

`lookupT key (Join left k v right)`

`| key == k = Just v`

`| key < k = lookupT key left`

`| key > k = lookupT key right`



Recursive type means
a recursive function!

Inserting a New Key

We also need a function to build tables. We define

$$\text{insertT} :: \text{Ord } k \Rightarrow k \rightarrow v \rightarrow \text{Table } k \ v \rightarrow \text{Table } k \ v$$

to insert a new key and value into a table.

We must be careful to insert the new entry in the right place, so that the keys remain in order.

Idea: Compare the new key against the middle one. Insert into the first or second half as appropriate.

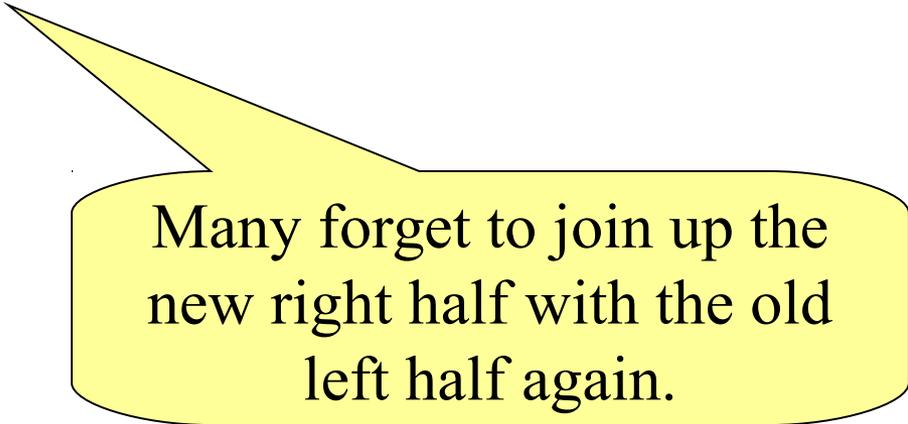
Defining Insert

`insertT key val Empty = Join Empty key val Empty`

`insertT key val (Join left k v right)`

`| key <= k = Join (insertT key val left) k v right`

`| key > k = Join left k v (insertT key val right)`



Many forget to join up the new right half with the old left half again.

Efficiency

On average, how many comparisons does it take to find a key in a table of 1000 entries, using a list and using the new method?

Using a list: 500

Using the new method: 10

Testing

- How should we test the Table operations?
 - By comparison with the list operations

```
prop_lookupT k t =  
  lookupT k t == lookup k (contents t)  
prop_insertT k v t =  
  contents (insertT k v t) == insert (k,v) (contents t)
```

contents :: Table k v -> [(k,v)]

Generating Random Tables

- Recursive types need recursive generators
instance (Arbitrary k, Arbitrary v) =>

Arbitrary (Table k v) **where**

We can generate arbitrary
Tables...

...provided we can generate
keys and values

Generating Random Tables

- Recursive types need recursive generators

instance (Arbitrary k, Arbitrary v) =>

Arbitrary (Table k v) **where**

arbitrary = oneof [return Empty,

do k <- arbitrary

v <- arbitrary

left <- arbitrary

right <- arbitrary

return (Join left k v right)]

Quiz:

What is wrong with
this generator?

Controlling the Size of Tables

- Generate tables with *at most n elements*

```
table s = frequency [(1, return Empty),  
                    (s, do k <- arbitrary  
                          v <- arbitrary  
                          l <- table (s `div` 2)  
                          r <- table (s `div` 2)  
                          return (Join l k v r))]
```

```
instance (Arbitrary k, Arbitrary v) =>  
          Arbitrary (Table k v) where  
    arbitrary = sized table
```

Testing Table Properties

```
prop_lookupT k t = lookupT k t == lookup k (contents t)
```

```
Main> quickCheck prop_lookupT
```

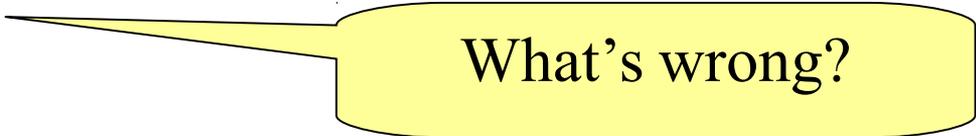
```
Falsifiable, after 10 tests:
```

```
0
```

```
Join Empty 2 (-2) (Join Empty 0 0 Empty)
```

```
Main> contents (Join Empty 2 (-2) ...)
```

```
[(2,-2),(0,0)]
```



What's wrong?

Tables must be Ordered!

```
prop_invTable :: Table Integer Integer -> Bool
prop_invTable t = ordered ks
  where ks = [k | (k,v) <- contents t]
```

- Tables should satisfy an important *invariant*.

```
Main> quickCheck prop_invTable
Falsifiable, after 4 tests:
Join Empty 3 3 (Join Empty 0 3 Empty)
```

How to Generate Ordered Tables?

- Generate a random list,
 - Take the *first* (key,value) to be at the root
 - Take all the *smaller* keys to go in the left subtree
 - Take all the *larger* keys to go in the right subtree

Converting a List to a Table

```
-- table kvs converts a list of key-value pairs into a Table
-- satisfying the ordering invariant
table :: Ord k => [(k,v)] -> Table k v
table []           = Empty
table ((k,v):kvs) = Join (table smaller) k v (table larger)
  where
    smaller = [(k',v') | (k',v') <- kvs, k' < k]
    larger  = [(k',v') | (k',v') <- kvs, k' > k]
```

Generating Ordered Tables

Keys must have an ordering

```
instance (Ord k, Arbitrary k, Arbitrary v) =>  
          Arbitrary (Table k v) where  
  arbitrary = do kvs <- arbitrary  
               return (table kvs)
```

List of keys
and values

Testing the Properties

- Now the invariant holds, but the properties don't!

```
Main> quickCheck prop_invTable
```

```
OK, passed 100 tests.
```

```
Main> quickCheck prop_lookupT
```

```
Falsifiable, after 7 tests:
```

```
-1
```

```
Join (Join Empty (-1) (-2) Empty) (-1) (-1) Empty
```

More Testing

```
prop_insertT k v t =  
  insert (k,v) (contents t)  
  == contents (insertT k v t)
```

```
Main> quickCheck prop_insertT  
Falsifiable, after 8 tests:  
0  
0  
Join Empty 0 (-1) Empty
```

What's
wrong?

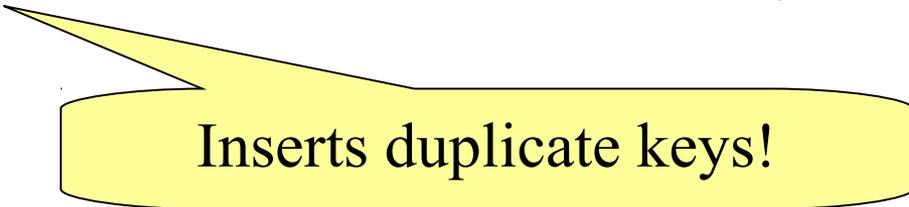
The Bug

`insertT key val Empty = Join Empty key val Empty`

`insertT key val (Join left k v right) =`

`| key <= k = Join (insertT key val left) k v right`

`| key > k = Join left k v (insertT key val right)`



Inserts duplicate keys!

The Fix

`insertT key val Empty = Join Empty key val Empty`

`insertT key val (Join left k v right) =`

`| key < k = Join (insertT key val left) k v right`

`| key == k = Join left k val right`

`| key > k = Join left k v (insertT key val right)`

```
prop_invTable :: Table Integer Integer -> Bool
```

```
prop_invTable tab = ordered ks && ks == nub ks
```

```
  where ks = [k | (k,v) <- contents tab]
```

(and fix the table generator)

Testing Again

```
Main> quickCheck prop_insertT
```

```
Falsifiable, after 6 tests:
```

```
-2
```

```
2
```

```
Join Empty (-2) 1 Empty
```

Testing Again

```
Main> quickCheck prop_insertT
```

```
Falsifiable, after 6 tests:
```

```
-2
```

```
2
```

```
Join Empty (-2) 1 Empty
```

```
Main> insertT (-2) 2 (Join Empty (-2) 1 Empty)
```

```
Join Empty (-2) 2 Empty
```

Testing Again

```
Main> quickCheck prop_insertT
```

```
Falsifiable, after 6 tests:
```

```
-2
```

```
2
```

```
Join Empty (-2) 1 Empty
```

```
Main> insertT (-2) 2 (Join Empty (-2) 1 Empty)
```

```
Join Empty (-2) 2 Empty
```

```
Main> insert (-2,2) [(-2,1)]
```

```
[(-2,1),(-2,2)]
```

insert doesn't *remove* the old key-value pair when keys clash – the wrong model!

Fixing prop_insertT

- Ad hoc fix:

```
prop_insertT k v t =  
  insert (k,v) [(k',v') | (k',v') <- contents t, k' /= k] ==  
  contents (insertT k v t)
```

Data.Map

- The standard module Data.Map contains an advanced tree-based implementation of tables

Summary

- Recursive datatypes can store data in different ways
- Clever choices of datatypes and algorithms can improve performance dramatically
- Careful thought about *invariants* is needed to get such algorithms right!
- Formulating properties and invariants, and testing them, reveals bugs early