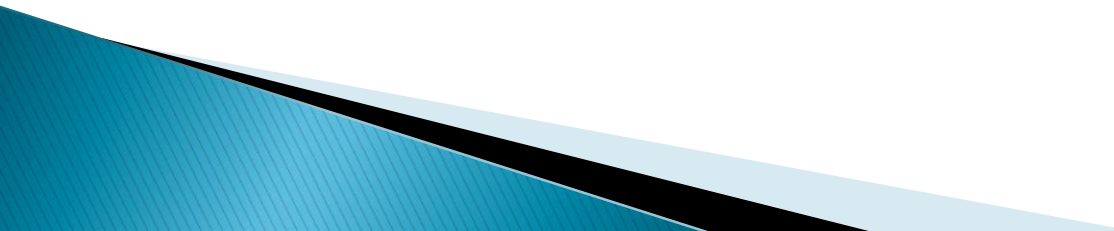


# Finger Trees

Edward Kmett

# Outline

- ▶ Performance
  - ▶ Trees
  - ▶ Fingers
  - ▶ Digits
  - ▶ Measures and Applications
  - ▶ Conclusion
- 

# What are Finger Trees?

- ▶ FingerTrees are a special form of “leafy” “monoidally annotated” 2–3 Tree that provide fast access from the first and last leaves.
- ▶ We call the first and last node in the tree the ‘fingers’ of the tree, in a sense we’ll make more rigorous shortly.

# Why FingerTrees? Performance

Operation	Amortized Bounds			
	Finger Tree	Annotated 2-3 Tree	List	Vector
cons, snoc	$O(1)$	$O(\log n)$	$O(1)/O(n)$	$O(n)$
viewl, viewr	$O(1)$	$O(\log n)$	$O(1)/O(n)$	$O(1)$
measure/length	$O(1)$	$O(1)$	$O(n)$	$O(1)$
append	$O(\log \min(\ell_1, \ell_2))$	$O(\log n)$	$O(n)$	$O(n+m)$
split	$O(\log \min(n, \ell-n))$	$O(\log n)$	$O(n)$	$O(1)$
replicate	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$
fromList, toList, reverse	$O(\ell)/O(\ell)/O(\ell)$	$O(\ell)$	$O(1)/O(1)/O(n)$	$O(n)$
index	$O(\log \min(n, \ell-n))$	$O(\log n)$	$O(n)$	$O(1)$

**FingerTrees are Fast!**

# Leafy and Traditional Binary Trees

- ▶ data Leafy a = Leaf a | Fork (Tree a) (Tree a)  
*or*
- ▶ data Tree a = Tip | Bin (Tree a) a (Tree a)

# Non-Empty Leafy Trees

- ▶ `data Leafy a = Leaf a | Fork (Leafy a) (Leafy a)`
- ▶ `instance Functor Leafy` where
  - `fmap f (Leaf a) = Leaf (f a)`
  - `fmap f (Fork l r) = Fork (fmap f l) (fmap f r)`
- ▶ `instance Monad Leafy` where
  - `return = Leaf`
  - `Leaf a >>= f = f a`
  - `Fork l r >>= f = Fork (l >>= f) (r >>= f)`

# Possibly-Empty Leafy Trees

- ▶ `data Leafy a = Leaf a | Fork (Leafy a) (Leafy a)`
- ▶ `data Tree a = Empty | Tree (Leafy a)`

`instance Functor Tree ...`

`instance Monad Tree ...`

# Traditional Trees

▶ `data Tree a = Tip | Bin (Tree a) a (Tree a)`

▶ `instance Functor Tree where`

`fmap _ Tip = Tip`

`fmap f (Bin l a r) = Bin (fmap f l) (f a) (fmap f r)`

▶ `instance Monad Tree where`

`return a = Bin Tip a Tip`

`Tip >>= _ = Tip`

`Bin l a r >>= f = (l >>= f) +++ f a +++ (r >>= f)`

`(+++) :: Tree a -> Tree a -> Tree a`

**A Rather More Expensive Monad**



# Traditional Trees (on Hackage)

- ▶ `data Tree a = Tip | Bin (Tree a) a (Tree a)`
- ▶ Lots of variations in the *containers* library

## Bounded Balance:

- ▶ `data Map k v = Tip | Bin !!Int !k v !(Map k v) !(Map k v)`
- ▶ `data Set k = Tip | Bin !!Int !k !(Set k) !(Set k)`

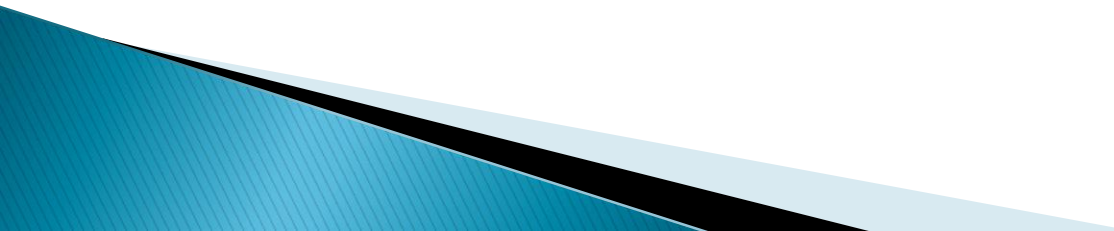
## PATRICIA Tries

- ▶ `data IntSet = Tip | Bin !!Int !IntSet !IntSet`
- ▶ `data IntMap v = Tip | Bin !!Int v !(IntMap v) !(IntMap v)`

# Differences

- ▶ `data Leafy a = Leaf a | Fork (Tree a) (Tree a)`
- ▶ `data Tree a = Tip | Bin (Tree a) a (Tree a)`

## Observations:

- ▶ Leafy trees are non-empty
  - ▶ You have to plod all the way down to the leaves to extract any values
  - ▶ They are a little easier to define as a monad
- 

# Similarities

```
data Node v a = Tip a | Bin (Tree v a) v (Tree v a)
```

- ▶ Both are special cases!

```
type Leafy a = Node () a
```

```
type Tree a = Node a ()
```

$v$  could be:

- ▶ a key that we use to walk the tree
- ▶ a summary of the leaves below it
- ▶ ...

# Similarities

```
data Tree v a = Empty | Tree (Node v a)
data Node v a = Tip a | Bin (Node v a) v (Node v a)
```

- ▶ Both are special cases!

```
type Leafy a = Node () a
type Tree a = Node a ()
```

$v$  could be:

- ▶ a key that we use to walk the tree
- ▶ a summary of the leaves below it
- ▶ ...

# How To Get Values for $v$

- ▶ We have a few options for how we to build trees.
- ▶ The containers library rather adequately covers most use cases that just store values in the nodes.

# Option 1: Annotated Trees as Monads

- ▶ `data Tree v a = Empty | Tree (Node v a)`
- ▶ `data Node v a = Tip a | Bin (Node v a) !v (Node v a)`
- ▶ `instance Functor (Node v) where`  
    `fmap f (Tip a) = Tip (f a)`  
    `fmap f (Bin l v r) = Bin (fmap f l) v (fmap f r)`
- ▶ `instance Monad (Node v) where`  
    `return = Tip`  
    `Tip a >>= f = f a`  
    `Bin l v r >>= f = Bin (l >>= f) v (r >>= f)`

`instance Functor (Tree v) ...`

`instance Monad (Tree v) ...`

**A Free Monad!**

# Option 1: Annotated Trees as Monads

- ▶ `data Tree v a = Empty | Tree (Node v a)`
- ▶ `data Node v a = Tip a | Bin (Node v a) !v (Node v a)`
- ▶ `instance Functor (Node v) where`  
    `fmap f (Tip a) = Tip (f a)`  
    `fmap f (Bin l v r) = Bin (fmap f l) v (fmap f r)`
- ▶ `instance Monad (Node v) where`  
    `return = Tip`  
    `Tip a >>= f = f a`  
    `Bin l v r >>= f = Bin (l >>= f) v (r >>= f)`

`instance Functor (Tree v) ...`

`instance Monad (Tree v) ...`

**Just Because Something Can Be a Monad  
Doesn't Mean It Should Be!**

# Option 1: Annotated Trees as Monads

- ▶ `data Tree v a = Empty | Tree (Node v a)`
- ▶ `data Node v a = Tip a | Bin (Node v a) !v (Node v a)`
- ▶ This monad and our lack of balancing are rather closely tied.
- ▶ You can't balance the tree because the number of leaves would change and the data is in the leaves.
- ▶ So while that monad is free, you get what you paid for.

**Just Because Something Can Be a Monad  
Doesn't Mean It Should Be!**



# Option 2: Monoidal Annotation

- ▶ class Monoid m where
  - mempty :: m
  - mappend :: m -> m -> m
  - mconcat :: [m] -> m
  - mconcat = foldr mappend mempty

( $\diamond$ ) = mappend -- (<>) is going into *base* !

Laws:

$$\forall a. \text{mempty} \diamond a = a \diamond \text{mempty} = a$$

$$\forall a b c. (a \diamond b) \diamond c = a \diamond (b \diamond c)$$

```
instance Monoid [] where
  mempty = []
  mappend = (++)
```

```
newtype Size = Size Int
  deriving (Show,Eq,Num)
```

```
instance Monoid Size where
  mempty = 0
  mappend = (+)
```

# Option 2: Monoidal Annotation

- ▶ `data Tree v a = Empty | Tree (Node v a)`
- ▶ `data Node v a = Tip a | Bin (Node v a) !v (Node v a)`
- ▶ `class Monoid v => Measured v a | a -> v where`
  - `measure :: a -> v`

```
bin :: Measured v a => Node v a -> Node v a -> Node v a
bin l r = Bin l (measure l `mappend` measure r) r
```


```
instance Measured v a => Measured v (Node v a) where
  measure (Tip a) = measure a
  measure (Bin _ v _) = v
```

```
instance Measured v a => Measured v (Tree v a) where
  measure Empty = mempty
  measure (Tree t) = measure t
```

# Trees as Monoids?

- ▶ `data Tree v a = Empty | Tree (Node v a)`
- ▶ `data Node v a = Tip a | Bin (Node v a) !v (Node v a)`
- ▶ `instance Measured v a => Monoid (Tree v a)` where
  - `mempty = Empty`
  - `Empty `mappend` r = r`
  - `l `mappend` Empty = l`
  - `Tree l `mappend` Tree r = bin l r`

# Trees as Monoids

- ▶ `data Tree v a = Empty | Tree (Node v a)`
- ▶ `data Node v a = Tip a | Bin (Node v a) !v (Node v a)`
- ▶ `instance Measured v a => Monoid (Tree v a)` where
  - `mempty = Empty`
  - `Empty `mappend` r = r`
  - `l `mappend` Empty = l`
  - `Tree l `mappend` Tree r = bin l r` 

We violate associativity! (even after we add balancing)

We need to be careful what knowledge we expose of our structure.

# Effective ML Haskell

“Make illegal states unrepresentable”

Yaron Minsky

<http://ocaml.janestcapital.com/?q=node/75>

# Effective ~~ML~~ Haskell

“Make illegal states unrepresentable”

Yaron Minsky

Let's express more invariants at the type level!

<http://ocaml.janestcapital.com/?q=node/75>

# A Starting Point: Complete Trees

▶ data Complete a = Nil | a :> Complete (a, a)

instance Functor Complete where

fmap \_ Nil = Nil

fmap f (a :> as) = f a :> fmap (both f) as where

both f (a,b) = (f a, f b)

left :: Complete a -> Maybe (Complete a)

left = *exercise*

# A Starting Point: Complete Trees

- ▶ data Complete a = Nil | a :> Complete (a, a)

Nil

1 :> Nil

2 :> (1,3) :> Nil

4 :> (2,6) :> ((1,3),(5,7)) :> Nil

...



# A Starting Point: Complete Trees

- ▶ data Complete a = Nil | a :> Complete (a, a)

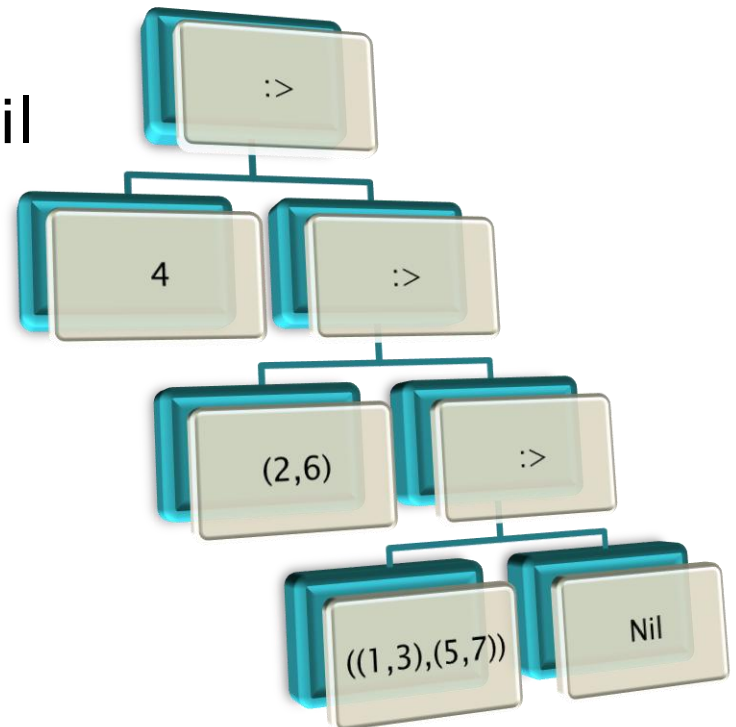
Nil

1 :> Nil

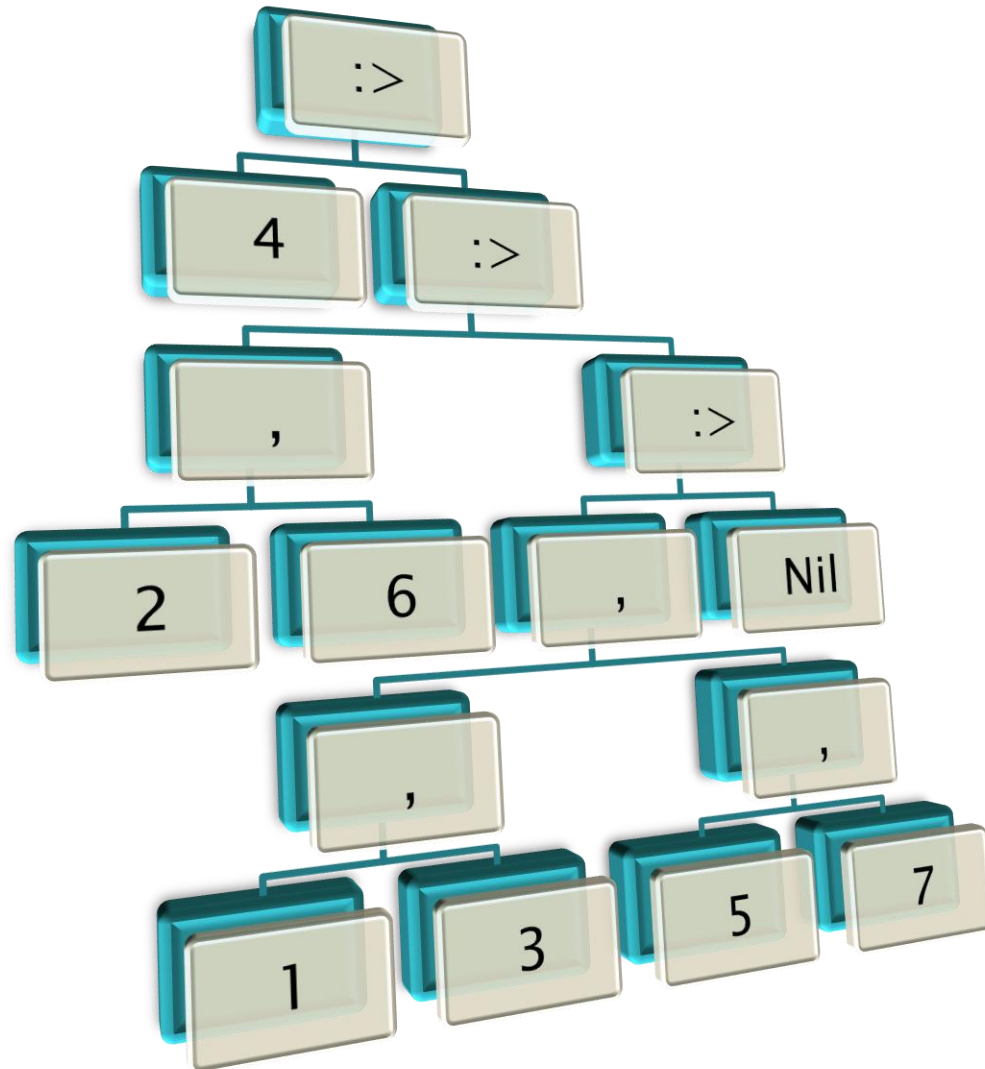
2 :> (1,3) :> Nil

4 :> (2,6) :> ((1,3),(5,7)) :> Nil

...



# A Starting Point: Complete Trees



# A Starting Point: Complete Trees

▶ data Complete a = Nil | a :> Complete (a, a)

instance Functor Complete where

fmap \_ Nil = Nil

fmap f (a :> as) = f a :> fmap (both f) as where

both f (a,b) = (f a, f b)

instance Applicative Complete where

pure a = a :> Nil

Nil <\*> \_ = Nil

\_ <\*> Nil = Nil

(f :> fs) <\*> (a :> as) = f a :> *exercise*

**Not a Monad**

# Complete Leafy Trees

- ▶ `data Square a = Zero a | Succ (Square (a, a))`
- ▶ `instance Functor Square` where
  - `fmap f (Zero a) = Zero (f a)`
  - `fmap f (Succ as) = Succ (fmap (both f) as)`

`instance Applicative Square` where

`pure = Zero`

`Zero f <*> a = fmap f a`

`fs <*> Zero a = fmap ($a) fs`

`Succ fs <*> Succ as = exercise`

**Not a Monad**

# Complete Leafy Trees

- ▶ `data Square a = Zero a | Succ (Square (a, a))`
- ▶ `instance Functor Square` where
  - `fmap f (Zero a) = Zero (f a)`
  - `fmap f (Succ as) = Succ (fmap (both f) as)`

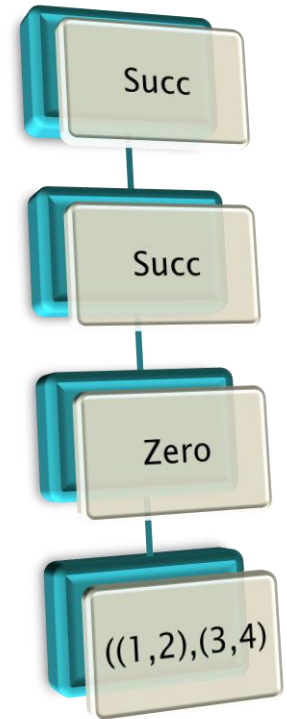
`instance Applicative Square` where

`pure = Zero`

`Zero f <*> a = fmap f a`

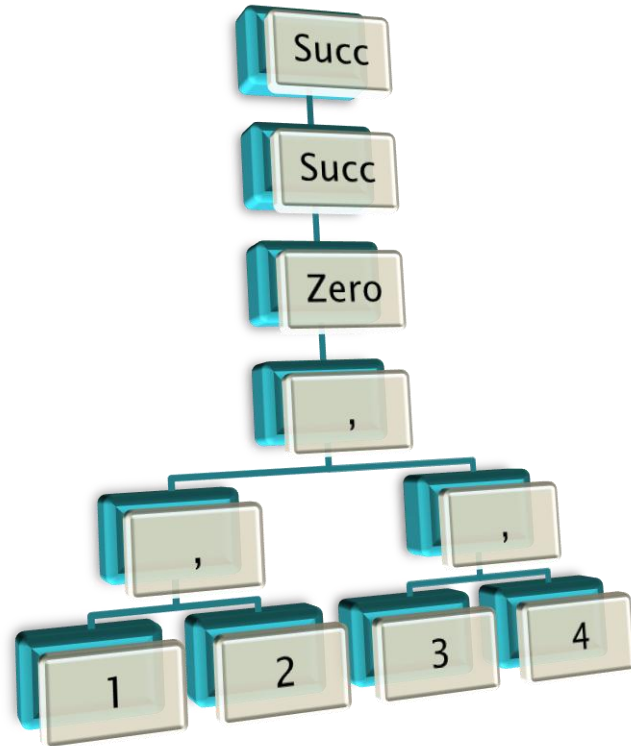
`fs <*> Zero a = fmap ($a) fs`

`Succ fs <*> Succ as = exercise`



# Complete Leafy Trees

- ▶ data Square a = Zero a | Succ (Square (a, a))



# Naïve 2-3 Trees

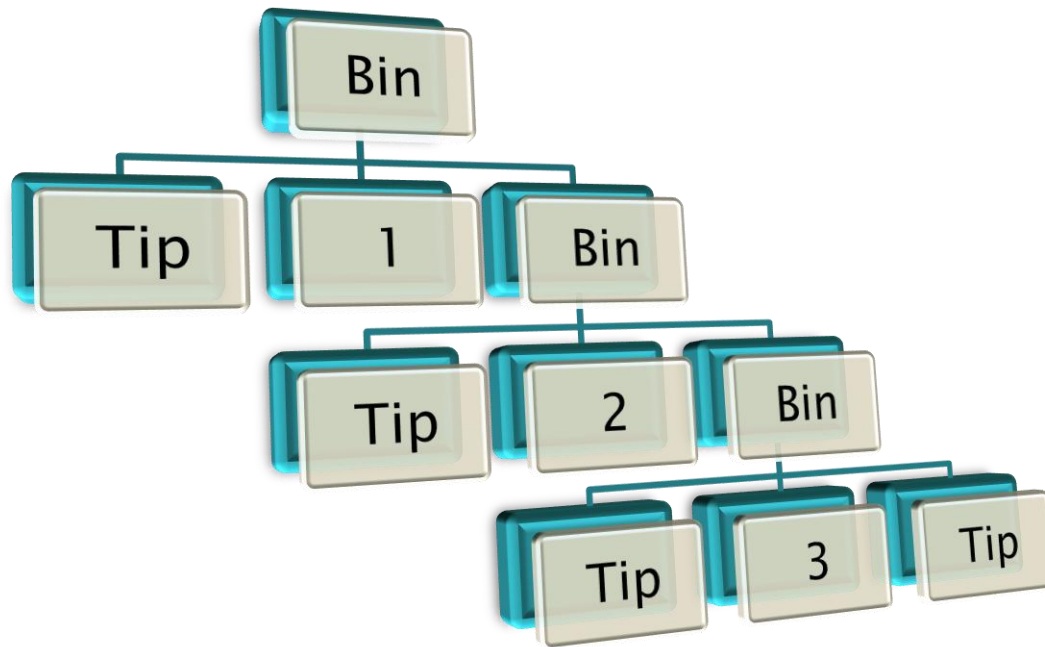
- ▶ data Tree v a =
  - Tip a
  - | Bin (Tree v a) v (Tree v a)
  - | Tri (Tree v a) v (Tree v a) v (Tree v a)

But again how do we know our tree is balanced?

Tip  
Bin Tip 1 Tip  
Tri Tip 1 Tip 2 Tip

Bin Tip 1 (Bin Tip 2 (Bin Tip 3 Tip))) -- we don't

# Naïve 2-3 Trees



Bin Tip 1 (Bin Tip 2 (Bin Tip 3 Tip)) -- we don't



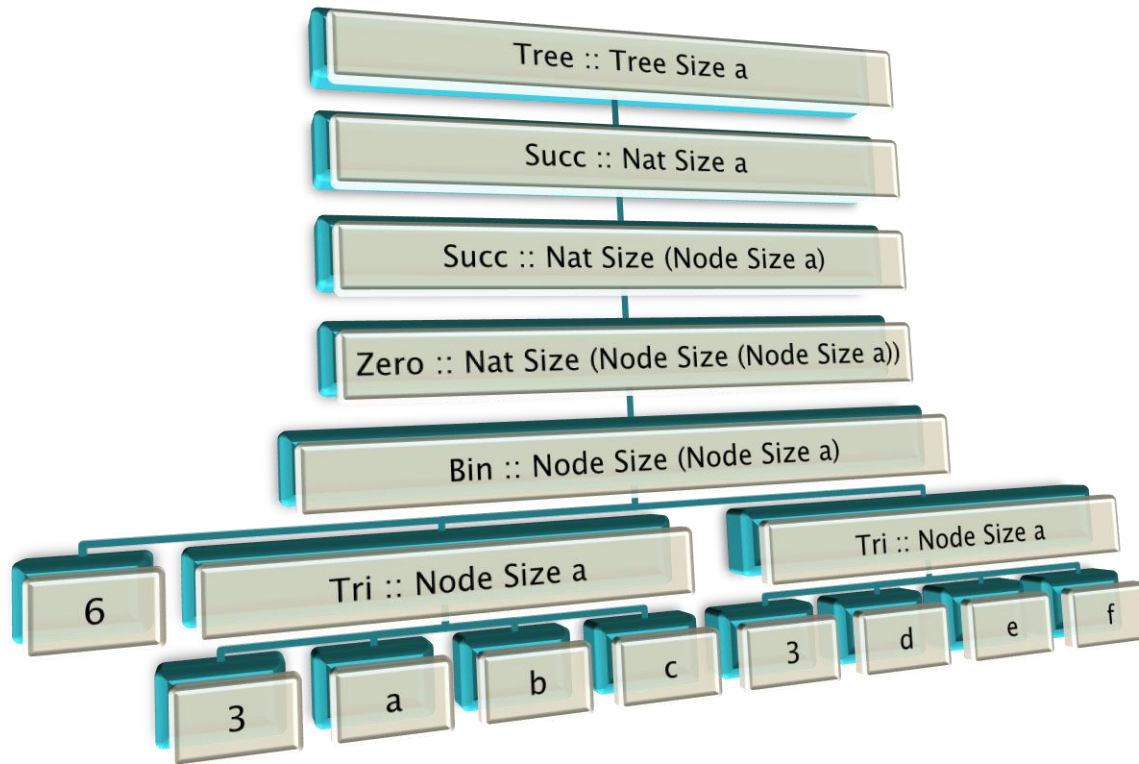
# Safe (Traditional) 2-3 Trees

- ▶ `data Tip a = Tip`
- ▶ `data Layer f a = Bin (f a) a (f a)`  
`| Tri (f a) a (f a) a (f a)`
- ▶ `data Nat f a = Zero (f a) | Succ (Nat (Layer f) a)`
- ▶ `newtype Tree a = Tree (Nat Tip a)`
  
- ▶ `Tree (Zero Tip)`
- ▶ `Tree (Succ (Zero (Bin Tip 1 Tip)))`
- ▶ `Tree (Succ (Zero (Tri Tip 1 Tip 2 Tip)))`
- ▶ `Tree (Succ (Succ (Zero (Bin (Bin Tip 1 Tip) 2 (Bin Tip 1 Tip))))))`
- ▶ As desired, `Bin Tip 1 (Bin Tip 2 (Bin Tip 3 Tip))` doesn't typecheck!

# Annotated 2-3 Trees

- ▶ `data Node v a = Bin !v a a | Tri !v a a a`
- ▶ `data Nat v a = Zero a | Succ (Nat v (Node v a))`
- ▶ `data Tree v a = Empty | Tree (Nat v a)`
  
- ▶ `Empty`
- ▶ `Tree (Zero a)`
- ▶ `Tree (Succ (Zero (Bin 2 a b)))`
- ▶ `Tree (Succ (Zero (Tri 3 a b c)))`
- ▶ `Tree (Succ (Succ (Zero (Bin 4 (Bin 2 a b) (Bin 2 c d)))))`
- ▶ `Tree (Succ (Succ (Zero (Bin 5 (Tri 3 a b c) (Bin 2 d e))))))`

# Annotated 2-3 Trees



Tree (Succ (Succ (Zero (Bin 6 (Tri 3 a b c) (Tri 3 d e f))))))

# Annotated 2-3 Trees

- ▶ `data Node v a = Bin !v a a | Tri !v a a a`
- ▶ `data Nat v a = Zero a | Succ (Nat v (Node v a))`
- ▶ `data Tree v a = Empty | Tree (Nat v a)`

# Annotated 2-3 Trees

- ▶ `newtype Elem a = Elem a`
- ▶ `data Node v a = Node2 !v a a | Node3 !v a a a`
- ▶ `data Nat v a = Zero a | Succ (Nat v (Node v a))`
- ▶ `data Tree v a = Empty | Tree (Nat v (Elem a))`
- ▶ This brings us into closer alignment with the notation in the Hinze and Paterson paper, and the 'Elem' newtype is useful to allow SPECIALIZE pragmas to fire discriminating between Node's and Elem's, allowing for better optimization.

# Intermission

- ▶ Coming Up:
  - Defining Fingers
  - Neat Finger Trees
  - Hinze and Paterson Finger Trees
  - Applications
  - Ropes

# Whence Fingers?

Fingers predate their use by [Hinze and Paterson](#). In fact they go back quite a ways:

D. Harel and G. Lueker. *A Data Structure with Movable Fingers and Deletions*. T.R. 145, Dept of ICS, University of California at Irvine, 1979.

They go back a couple years farther than that, but I haven't tracked down the original paper!



# What are Fingers?

- ▶ Finger are fast access points.
- ▶ A notable example with a single movable finger is a Zipper:
- ▶ `data ListZipper a = LZ ![a] [a]`
- ▶ `empty = LZ [] []` -- O(1)
- ▶ `insert x (LZ path xs) = LZ (path (x:xs))` -- O(1)
- ▶ `adjust f (LZ path (x:xs)) = LZ (path (f x:xs))` -- O(1)
- ▶ `delete (LZ path r) = LZ path (tail r)` -- O(1)
- ▶ `left (LZ (x:path) xs) = LZ path (x:xs)` -- O(1)
- ▶ `right (LZ path (x:xs)) = LZ (x:path) xs` -- O(1)
- ▶ `toList (LZ path xs) = reverse path ++ xs` -- O(n)
- ▶ `fromList = LZ []` -- O(1)

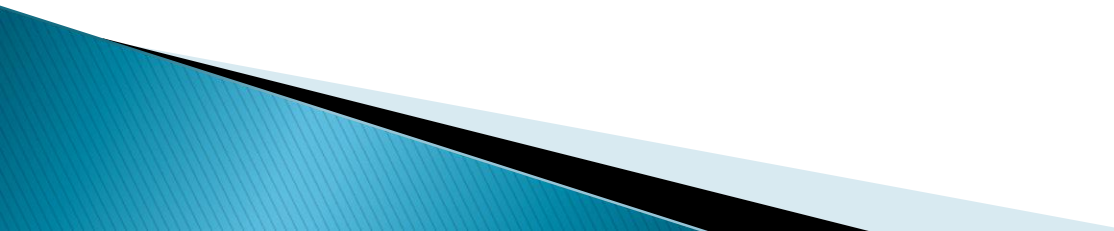


# Finger Search Trees

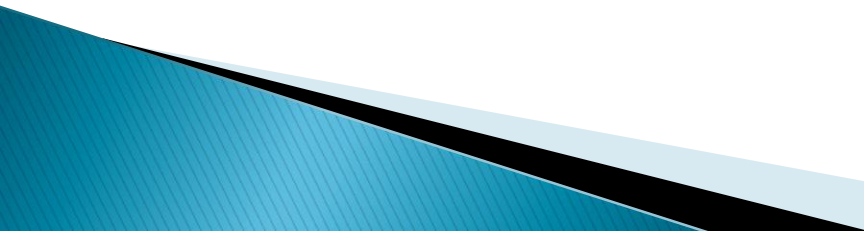
- ▶ Finger Search Tree's were designed to allow multiple such fast movable access points and a number of operations.
- ▶ In a transient imperative setting you can support an unbounded number of fingers, and obtain even nicer asymptotics. [[Brodal et al.](#)]
- ▶ Splay trees provide the same  $O(\log \text{ distance to finger})$  bound as a one finger tree. [[Cole](#)]

# Imperative Fingers

Shamelessly stolen from a blog post reply by David Eppstein:

- ▶ **empty**: an empty finger search tree. This doubles as a finger.
  - ▶ **find**  $f$   $x$ : find value  $x$  starting from finger  $f$ ; return a finger to  $x$ 's location.  $f$  remains a valid finger.
  - ▶ **delete**  $f$ : delete the value pointed to by finger  $f$ , return a finger to the element just larger than the one at  $f$ .  $f$  is no longer a valid finger.
  - ▶ **insert**  $f$   $x$ : insert a value  $x$  to the right of finger  $f$ , returning a finger pointing at  $x$ .  $f$  remains a valid finger.
- 

# In Search of a Swiss Army Knife

- Annotated 2–3 Trees make a half-way decent general purpose data structure once you get good at picking monoids, but you pick up an unfortunately logarithmic factor on most accesses.
  - The left- and right-most nodes in a tree do a lot of work when trying to use a tree as a deque
  - H&P's first idea: let the user emulate  $n$  fingers by juggling multiple trees as long as you have a tree that supports a finger to its left- and right-most nodes, and we can implement that functionally.
  - H&P's second idea: You can fix the remaining asymptotic issues by allowing a bit of slop on the edges.
- 

# “Neat” Finger Trees

```
data Node v a = Bin !v a a | Tri !v a a a
```

```
data Tree v a = E | S a | N !(Node v a) |
```

```
  | D !v !(Node v a) !(Tree v (Node v a)) !(Node v a)
```

# “Neat” Finger Trees

```
data Node v a = Bin !v a a | Tri !v a a a
data Tree v a = E | S a | N !(Node v a) |
                | D !v !(Node v a) !(Tree v (Node v a)) !(Node v a)
```

## Size-Annotated Tree => Neat Finger Tree

Empty =>

E

Tree (Zero z) =>

S z

Tree (Succ (Zero (Bin 2 y z))) =>

N (Bin 2 y z)

Tree (Succ (Succ (Zero (Bin 4 (Bin 2 w x) (Bin 2 y z)))))) =>

D 4 (Bin 2 w x) E (Bin 2 y z)

Tree (Succ (Succ (Zero (Tri 6 (Bin 3 a b) (Bin 2 c d) (Bin 2 e f)))))) =>

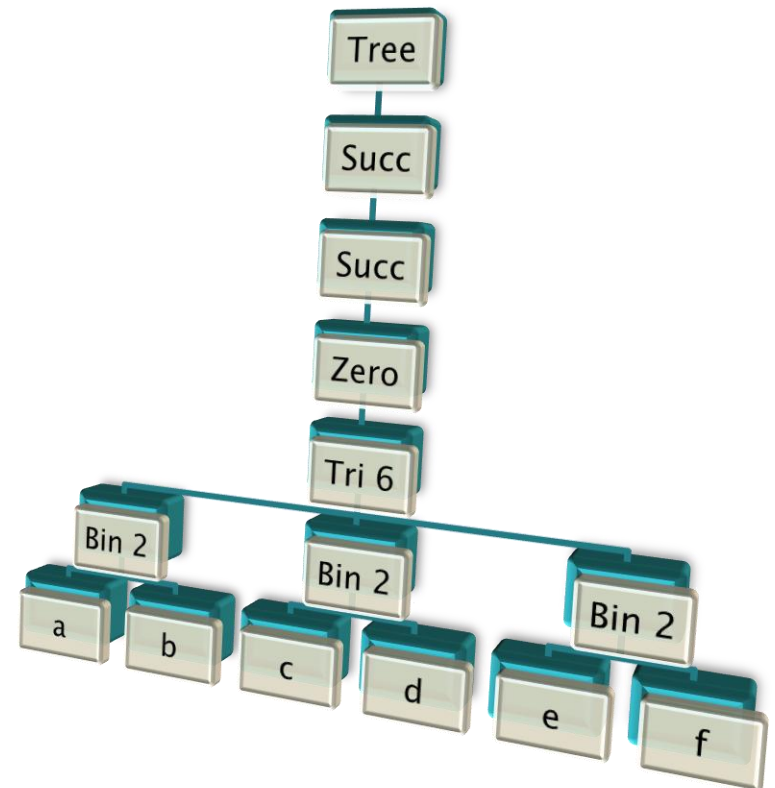
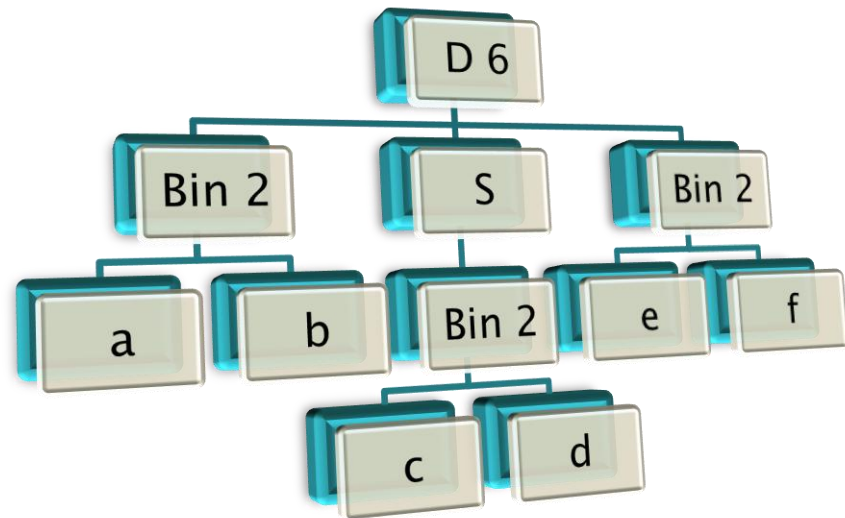
D 6 (Bin 2 a b) (N (Bin 2 c d)) (Bin 2 e f)

# “Neat” Finger Trees

data Node v a = Bin !v a a | Tri !v a a a

data Tree v a = E | S a | N !(Node v a) |

| D !v !(Node v a) (Tree v (Node v a)) !(Node v a)

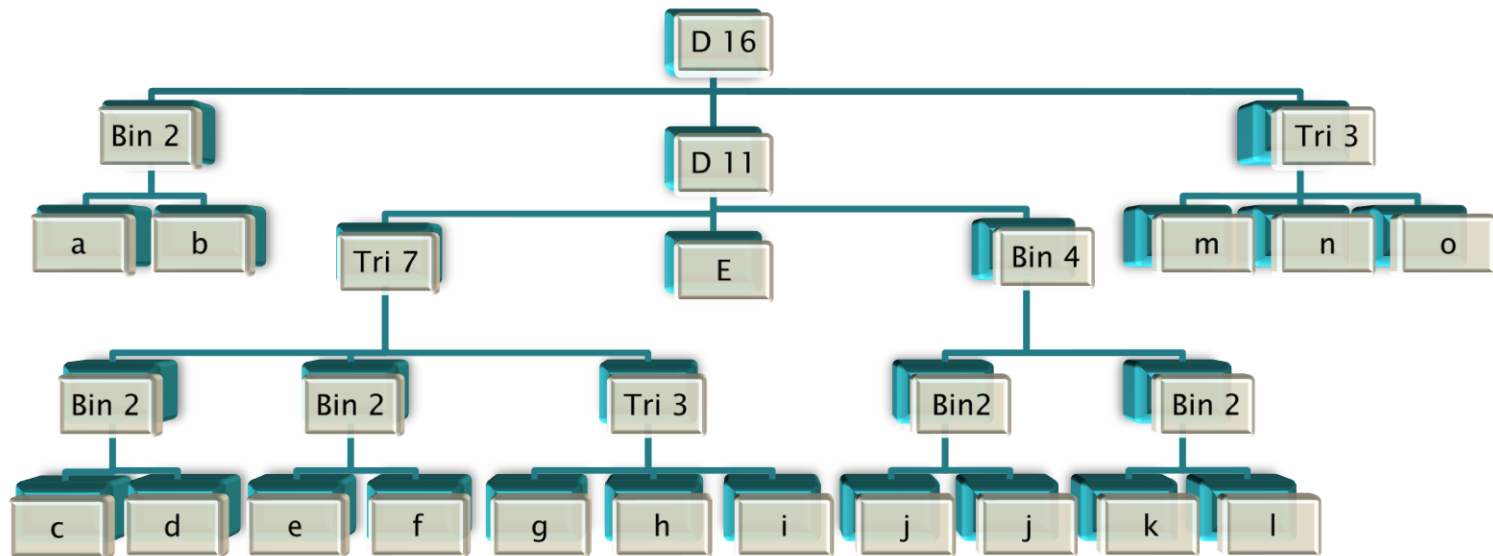


# “Neat” Finger Trees

```
data Node v a = Bin !v a a | Tri !v a a a
```

```
data Tree v a = E | S a | N !(Node v a) |
```

```
  | D !v !(Node v a) !(Tree v (Node v a)) !(Node v a)
```



# “Neat” Finger Trees

```
data Node v a = Bin !v a a | Tri !v a a a
data Tree v a = E | S a | N !(Node v a) |
                | D !v !(Node v a) !(Tree v (Node v a)) !(Node v a)
```

Sadly the asymptotics on “Neat” FingerTrees aren’t as good as they could be.

cons’ing can be  $O(\log n)$  and viewing are still  $O(\log n)$ !



# H&P Finger Trees

```
data Digit a = One a | Two a a | Three a a a | Four a a a a
data Node v a = Node2 !v a a | Node3 !v a a a
data FingerTree v a = Empty | Single a
  | Deep !v !(Digit a) !(FingerTree v (Node v a)) !(Digit a)
```

Empty

Single 'z'

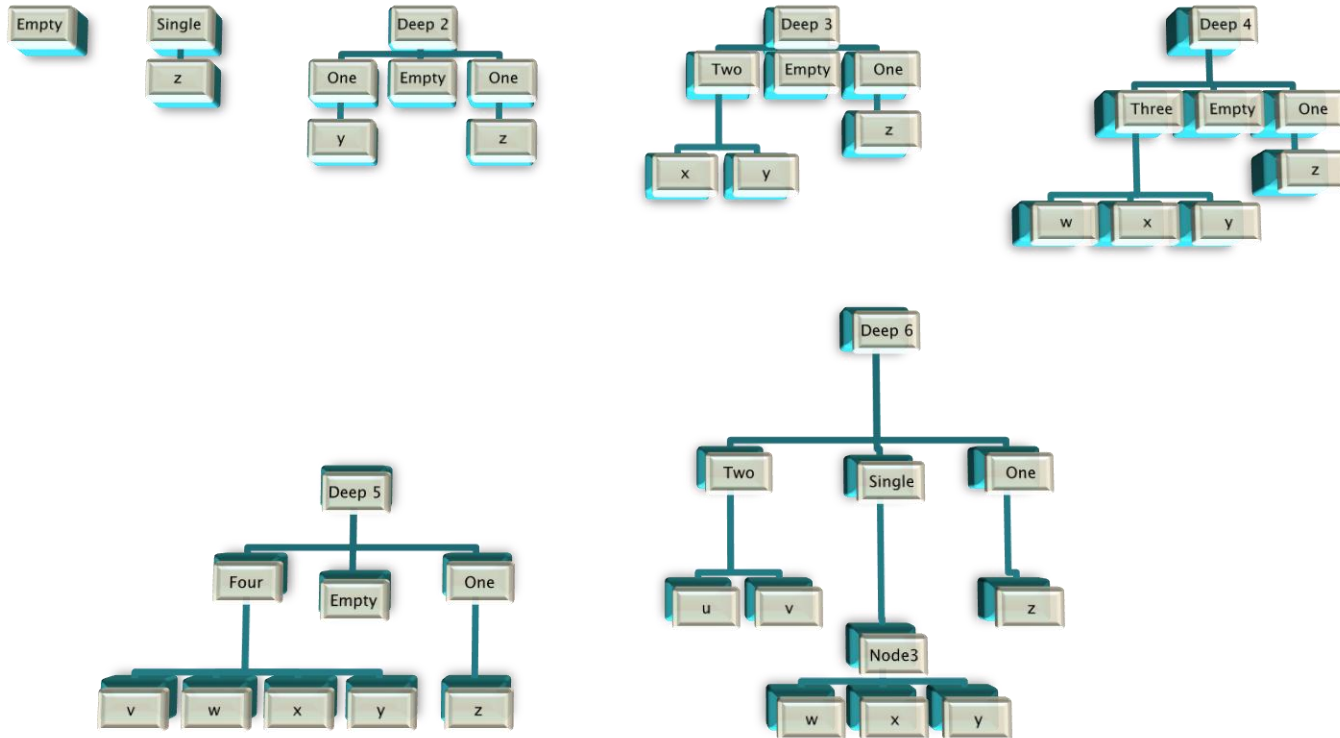
Deep 2 (One 'y') Empty (One 'z')

Deep 5 (Four 'v' 'w' 'x' 'y') Empty (One 'z')

Deep 7 (Four 't' 'u' 'v' 'w') (Single (Node2 2 'x' 'y')) (One 'z')

# A Rogue's Gallery

.. <| w <| x <| y <| z <| empty



# H&P Finger Trees

```
data Digit a = One a | Two a a | Three a a a | Four a a a a
data Node v a = Node2 !v a a | Node3 !v a a a
data FingerTree v a = Empty | Single a
  | Deep !v !(Digit a) !(FingerTree v (Node v a)) !(Digit a)
```

```
(<|) :: Measured v a => a -> FingerTree v a -> FingerTree v a
a <| Empty = Single a
a <| Single b = deep (One a) Empty (One b)
a <| Deep (One b) m sf = deep (Two a b) m sf
a <| Deep (Two b c) m sf = deep (Three a b c) m sf
a <| Deep (Three b c d) m sf = deep (Four a b c d) m sf
a <| Deep (Four b c d e) m sf =
  deep (Two a b) (node3 c d e <| m) sf
```

# H&P Finger Trees

```
data Digit a = One a | Two a a | Three a a a | Four a a a a
data Node v a = Node2 !v a a | Node3 !v a a a
data FingerTree v a = Empty | Single a
    | Deep !v !(Digit a) !(FingerTree v (Node v a)) !(Digit a)
data ViewL s a = EmptyL | a :< s a
```

```
viewl :: Measured v a => FingerTree v a -> ViewL (FingerTree v) a
viewl Empty = EmptyL
viewl (Single x) = x :< Empty
viewl (Deep _ (One x) m sf) = x :< case viewl m of
    EmptyL -> digitToTree sf
    a :< m' -> deep (nodeToDigit a) m' sf
viewl (Deep _ pr m sf) = lheadDigit pr :< deep (ltailDigit pr) m sf
```

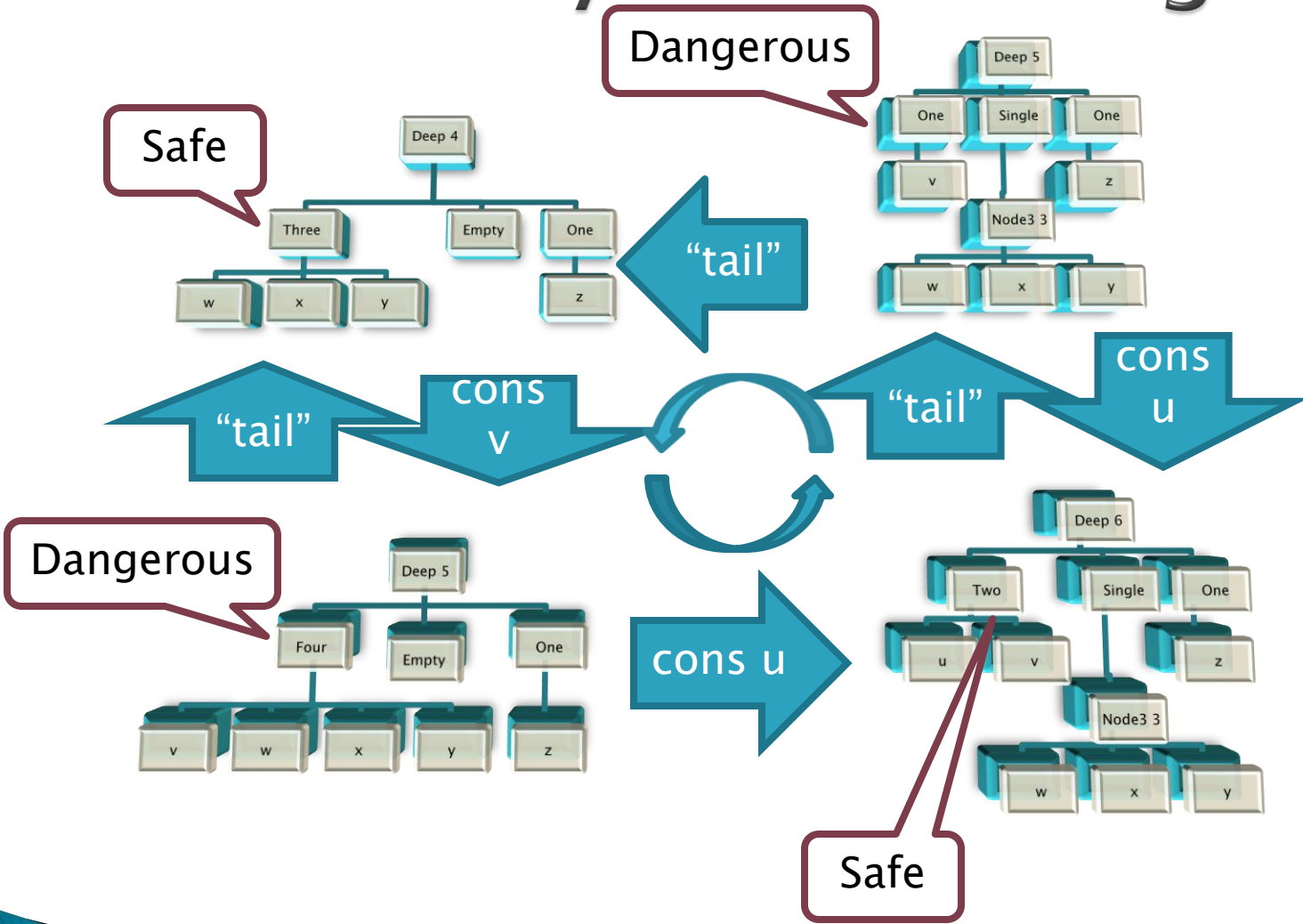
# H&P Finger Trees

```
data Digit a = One a | Two a a | Three a a a | Four a a a a
data Node v a = Node2 !v a a | Node3 !v a a a
data FingerTree v a = Empty | Single a
  | Deep !v !(Digit a) !(FingerTree v (Node v a)) !(Digit a)
```

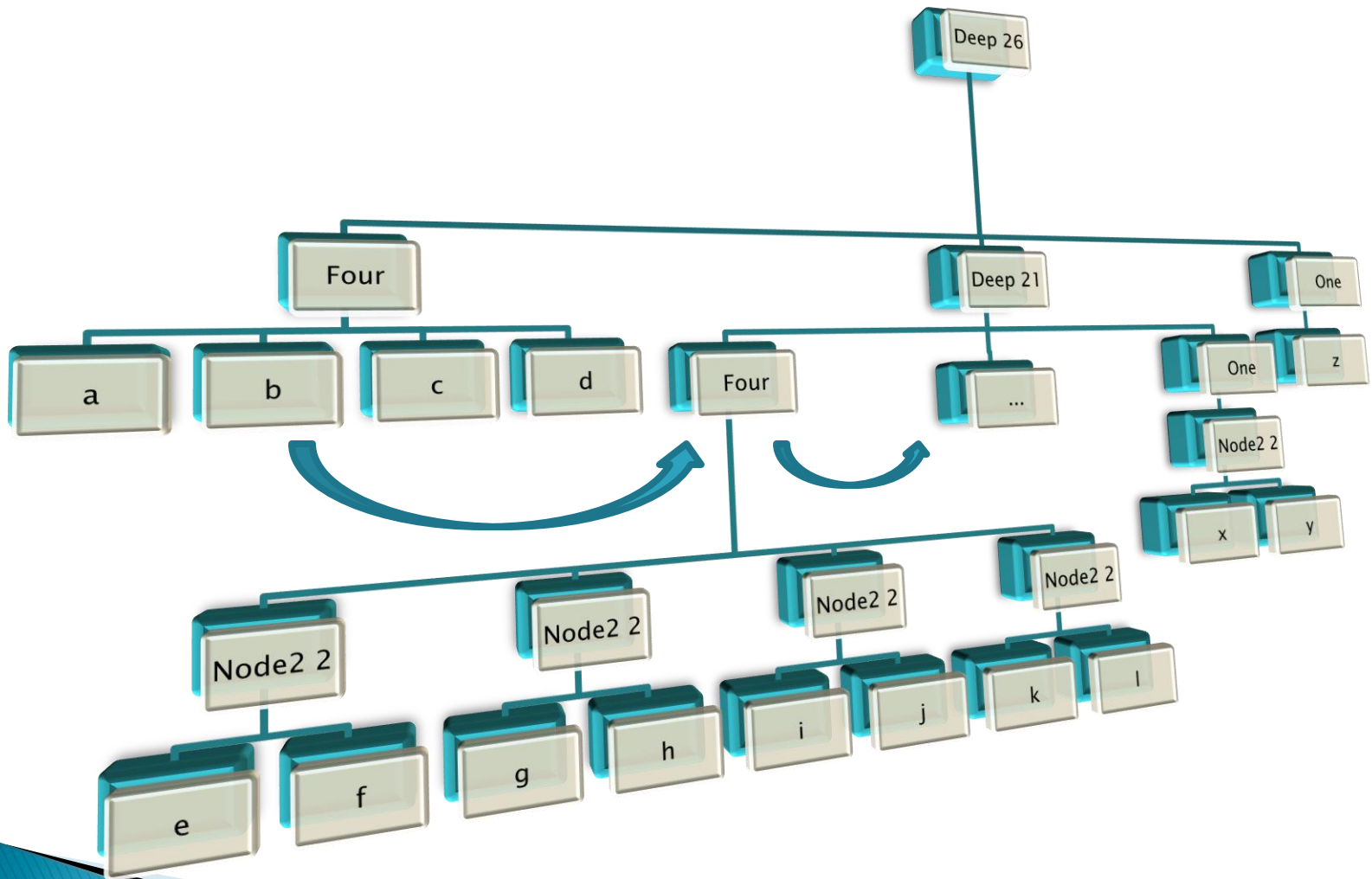
## Observations:

- ▶ Digit and Single don't bother to track the measure, assuming the monoid is inexpensive to recalculate.
- ▶ Digits can contain 1–4 elements, allowing for the aforementioned 'slop'

# Redundancy and Slowing Down



# Redundancy and Slowing Down



# Split

```
split :: Measured v a =>  
      (v -> Bool) ->  
      Fingertree v a ->  
      (FingerTree v a, FingerTree v a)
```

split breaks a fingertree on some location where the predicate applied to the value of the measure of the prefix of the fingertree, changes from false to true.

using the Size monoid:

```
split (> 2) (Deep 5 (Four a b c d) Empty (One e) =  
(Deep 2 (One a) Empty (One b), Deep (Two c d) Empty (One e))
```



# Applications

## Killer App:

Sequences, Data.Sequence is even Haskell 98!

## But Size isn't the only Monoid!

- ▶ Priority Queues
- ▶ Ordered Sequences
- ▶ Ropes
- ▶ Interval Trees
- ▶ Tracking Line Numbers in Source Code
- ▶ ...

**All You Need is the Right Monoid!**

# Ordered Sequences

```
newtype Elem a = Elem a
```

```
data Key a = NoKey | Key a deriving (Ord)
```

```
data OrdSeq a = FingerTree (Key a) (Elem a)
```

```
instance Monoid (Key a) where
```

```
  mempty = NoKey
```

```
  k `mappend` NoKey = k
```

```
  _ `mappend` k = k
```

```
instance Measured (Key a) (Elem a) where
```

```
  measure (Elem x) = Key x
```

# Ordered Sequences

```
newtype Elem a = Elem a
```

```
data Key a = NoKey | Key a
  deriving (Ord)
```

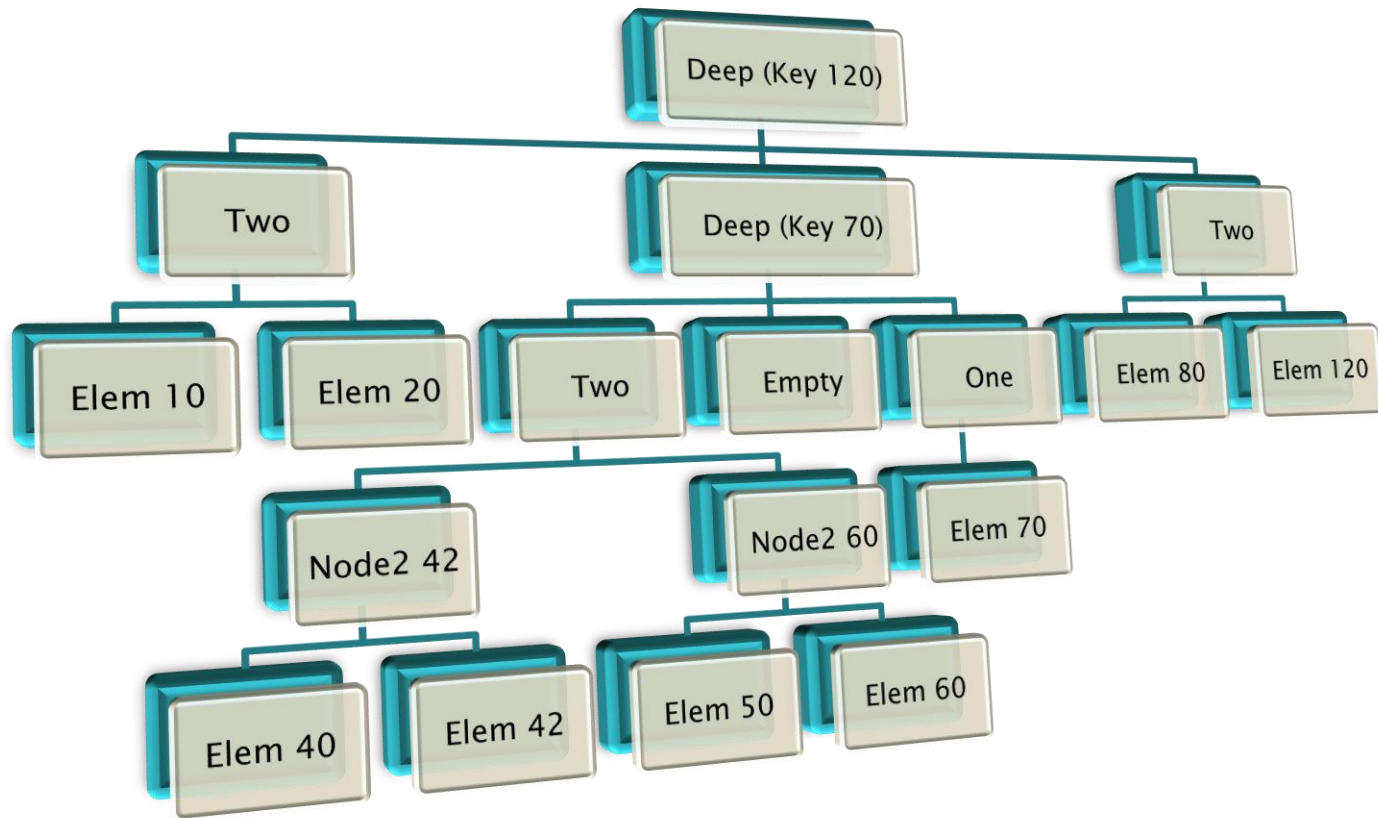
```
data OrdSeq a = FingerTree (Key a) (Elem a)
```

```
partition :: Ord a => a -> OrdSeq a -> (OrdSeq a, OrdSeq a)
partition k = split (>= Key k)
```

```
insert :: Ord a => a -> OrdSeq a -> OrdSeq a
insert x xs = l >< (Elem x <| r)
  where (l,r) = partition x xs
```

- ▶ ( $><$ ) - is a FingerTree 'append'

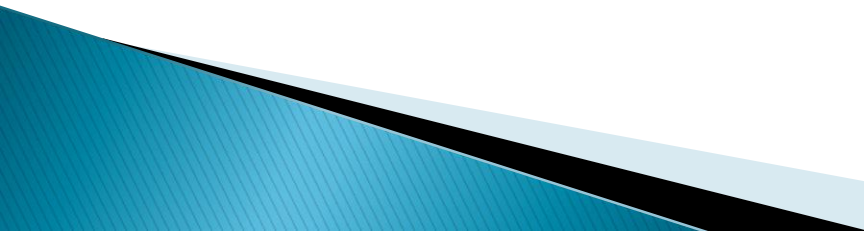
# Ordered Sequences



# Priority Queues

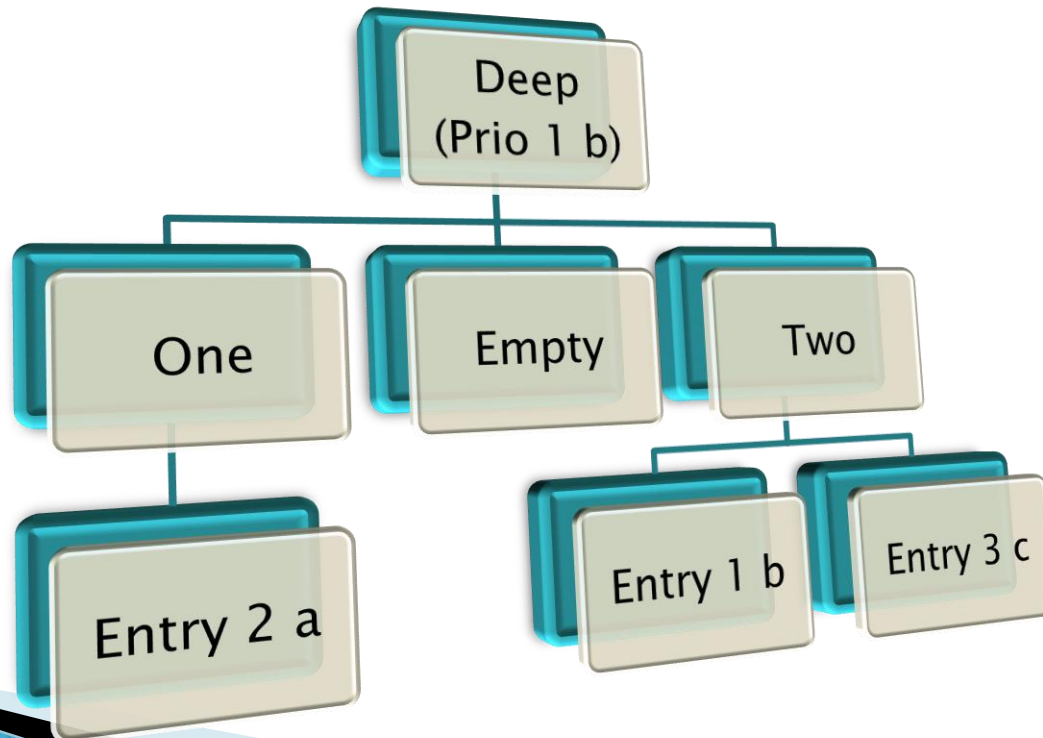
- ▶ `data Entry k v = Entry k v`
- ▶ `data Prio k v = Infinity | Prio k v` – unbounded min monoid
- ▶ `type PQ k v = FingerTree (Prio k v) (Entry k v)`
  
- ▶ `instance Monoid (Prio k v)` where
  - `mempty = Infinity`
  - `x `mappend` Infinity = x`  
`Infinity `mappend` y = y`  
`x@(Prio kx _) `mappend` y@(Prio ky _)`
    - | `kx <= ky = x`
    - | `otherwise = y`
  
- ▶ `instance Measured (Prio k v) (Entry k v)`
  - `measure (Entry k v) = Prio k v`
  
- ▶ `insert k v q = Entry k v <| q`
- ▶ `union q q' = q >< q'`

# Priority Queues

- ▶ data Entry k v = Entry k v
  - ▶ data Prio k v = Infinity | Prio k v
  - ▶ type PQ k v = FingerTree (Prio k v) (Entry k v)
  
  - ▶ dequeue :: Ord k => PQ k v -> Maybe (k, v, PQ k v)
  - ▶ dequeue q
    - | null q = Nothing
    - | otherwise = Just (k, v, case viewl t of \_ :< r' -> l >< r')
- where
- Prio k v = measure q
  - (l, r) = split belowk q
  - belowk Infinity = False
  - belowk (Prio k' \_) = k' <= k
- 

# Priority Queues

- ▶ data Entry k v = Entry k v
- ▶ data Prio k v = Infinity | Prio k v
- ▶ type PQ k v = FingerTree (Prio k v) (Entry k v)



# Priority Queues

**dequeue:**

take measure

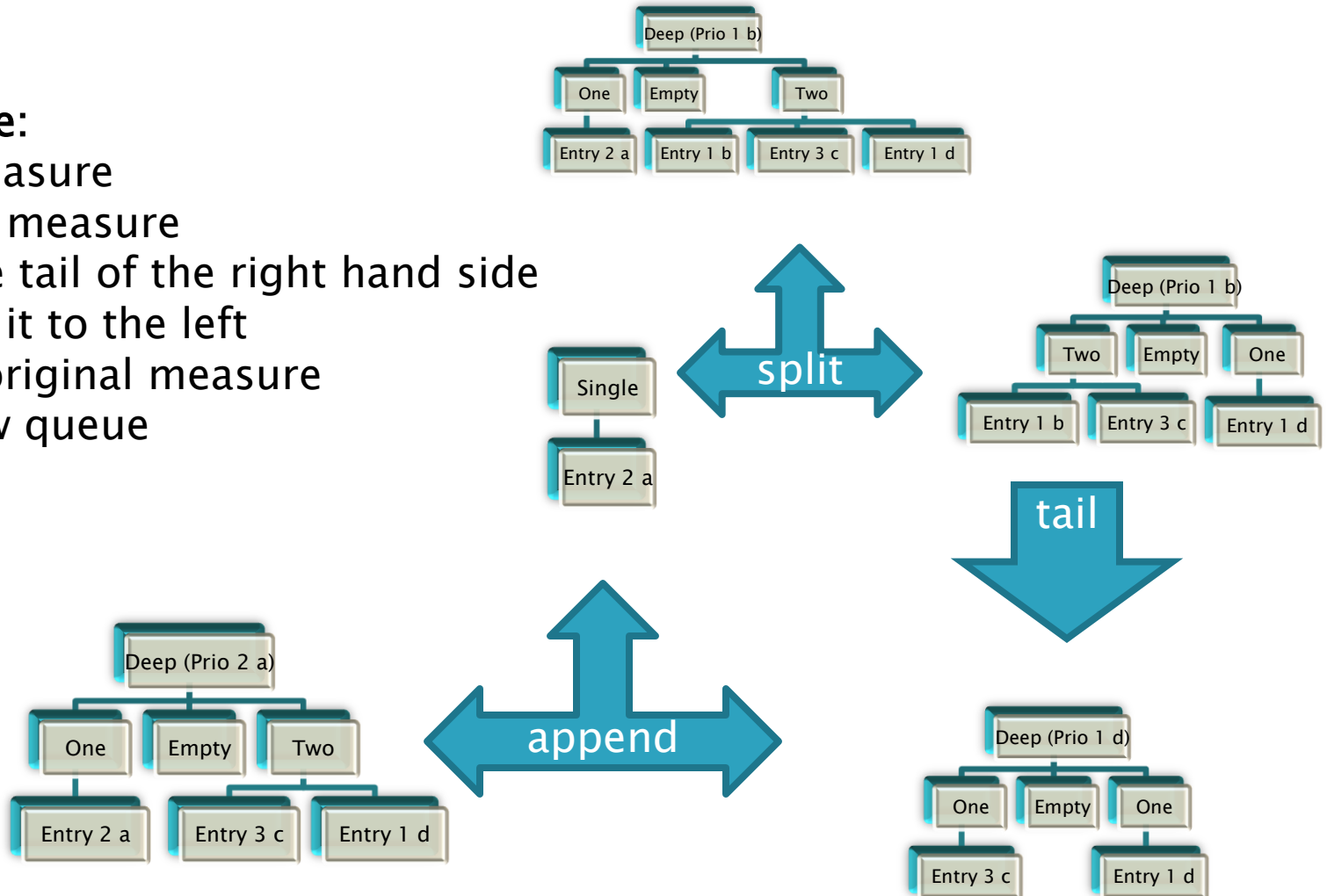
split on measure

take the tail of the right hand side

append it to the left

return original measure

and new queue





# Priority Queues

- ▶ Asymptotics Don't Say Everything
  - FingerTree Based Priority Queues can be  $\sim 10x$  Slower than a skew heap-based Priority queue traditional Priority Queue.
  - But it is stable so it can be used for fair scheduling) and has other nice properties.

# Designing Your Monoid

- ▶ A product of two monoids is a monoid

instance (Monoid a, Monoid b) => Monoid (a,b)

Last a, Sum a, Size, Any, All, even FingerTree v a are all good building blocks!

Design for the use of split:

Ideally you want to be able to ask a monotone increasing question using split. If your function doesn't go from False to True and stay True over the course of accumulating your monoid left to right over the values in your tree, you may not get an answer!

# Ropes

- ▶ Data.Sequence has awesome asymptotics
  - But it uses a lot of space and has some bad constant factors.

Data.Vector has great asymptotics, and nice fusion properties, but has  $O(n+m)$  append.

Rough Concept:

```
type Rope a = FingerTree Size (Vector a)
instance Measured Size (Vector a) where
    measure = Size . length
```

See *ropes* on hackage -- my implementation there uses ByteString.

# Why Ropes? Performance

Operation	Amortized Bounds		
	Finger Tree	Vector	Rope
▶ cons, snoc:	$O(1)$	$O(n)$	$O(1)$
▶ append:	$O(\log \min(\ell_1, \ell_2))$	$O(\ell_1 + \ell_2)$	$O(\log \min(\ell_1 / c, \ell_2 / c))$
▶ split:	$O(\log \min(n, \ell - n))$	$O(1)$	$O(\log \min(n / c, (\ell - n) / c))$ *
▶ replicate	$O(\log n)$	$O(n)$	$O(\log n)$
▶ index	$O(\log \min(n, \ell - n))$	$O(1)$	$O(\log \min(n / c, (\ell - n) / c))$

While the asymptotics remain the same, the constant space and access times reduce dramatically.  $c$  is between 32 and  $\ell$ , depending on how the rope is constructed.

Furthermore the elements of the rope can be stored unboxed, so the garbage collector doesn't spend all of its time inspecting every node in your Seq or String!

\* Also note, that unless you use append, the complexity of split remains  $O(1)$ . ;)

Dramatic Wins:

Space Consumption (0.7–30x better)

Garbage Collection Time (same with any unboxed primitive)

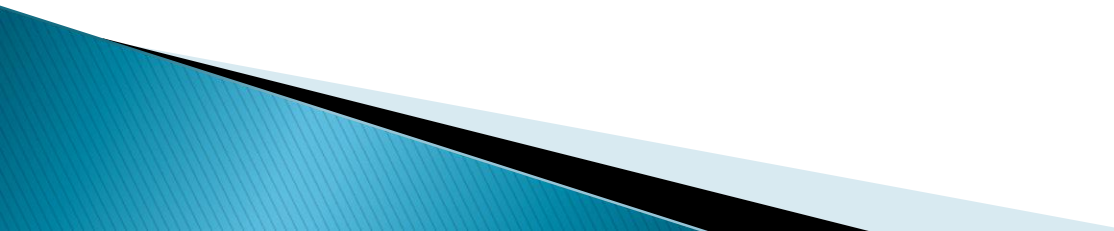
Speed (needs benchmarks)

**FingerTrees are Fast!**

# Conclusion

I hope that I have motivated how FingerTrees work and highlighted some interesting applications.

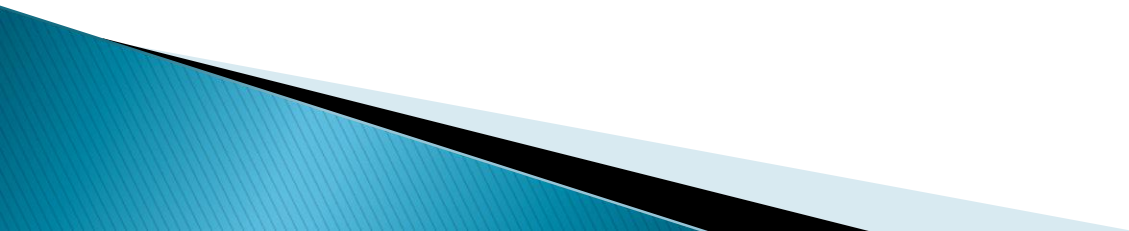
The main areas where FingerTrees shine is in their robust asymptotic performance under a wide array of usage scenarios. FingerTrees make a compelling case for being functional programming's "swiss army knife".



**Any Questions?**



# Extra Slides



# Chris Okasaki Red-Black Trees

- ▶ data Unit a = E
- ▶ data Red f a = C (f a) | R (f a) a (f a)
- ▶ data Layer f a = B (Red f a) a (Red f a)
- ▶ data RB f a = Z (f a) | S (RB (Layer f) a)
- ▶ type Tree = RB Unit

(A Non-Leafy Red-Black Tree)