Fortgeschrittene Funktionale Programmierung in Haskell

Universität Bielefeld, Sommersemester 2015

Jonas Betzendahl & Stefan Dresselhaus

Übersicht I

Worum soll es heute gehen?

- Funktionale Programmierung generell

Worum soll es heute gehen?

- Funktionale Programmierung generell
- Implementierung einer Double-Linked-List
  Wie macht man sowas in Haskell?

Worum soll es heute gehen?

- Funktionale Programmierung generell
- Implementierung einer Double-Linked-List
  Wie macht man sowas in Haskell?
- Lazyness
  Was für Auswirkungen hat das auf die Programmierung?
  Was für Möglichkeiten bietet dies?

Eine Double-Linked-List ist die klassische Einstiegs-Datenstruktur in der imperativen Welt.

Eine Double-Linked-List ist die klassische Einstiegs-Datenstruktur in der imperativen Welt.
Sie besteht aus

- Einem Paar von Pointern, die auf den Anfang und das Ende zeigen

Eine Double-Linked-List ist die klassische Einstiegs-Datenstruktur in der imperativen Welt.
Sie besteht aus

- Einem Paar von Pointern, die auf den Anfang und das Ende zeigen
- Aus Elementen, welche bestehen aus

Eine Double-Linked-List ist die klassische Einstiegs-Datenstruktur in der imperativen Welt.
Sie besteht aus

- Einem Paar von Pointern, die auf den Anfang und das Ende zeigen
- Aus Elementen, welche bestehen aus
  - Einem Pointer auf das nächste Element (null, falls nicht da)

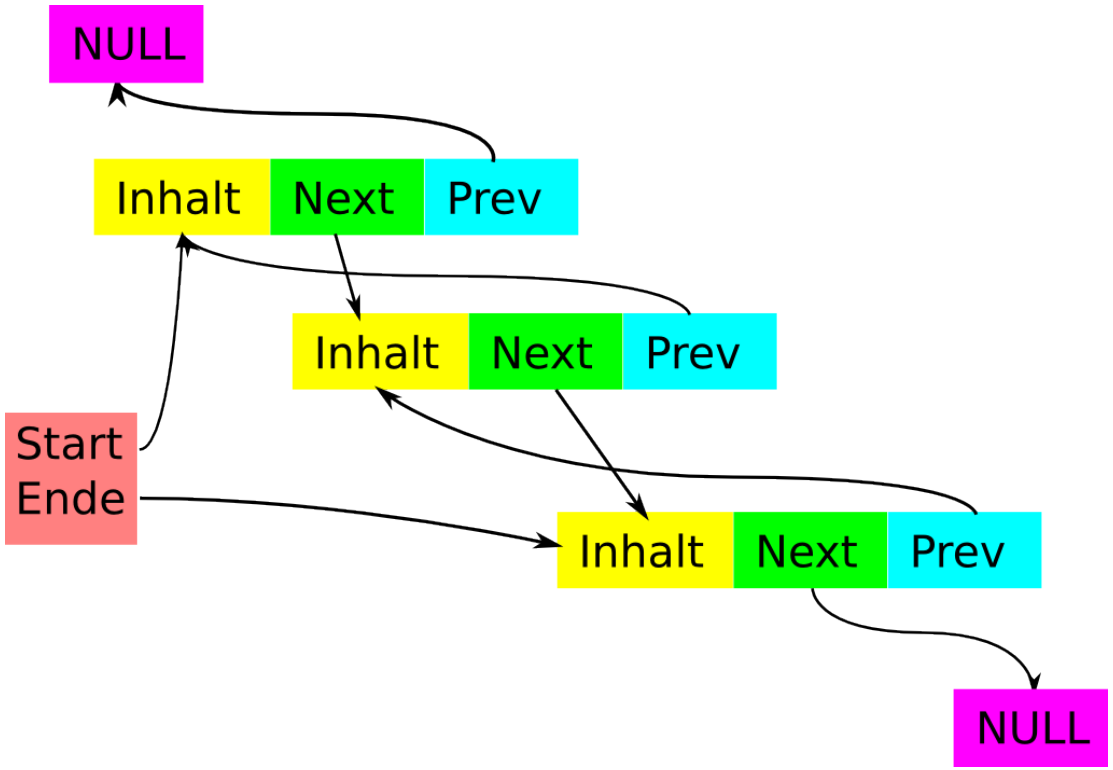
Eine Double-Linked-List ist die klassische Einstiegs-Datenstruktur in der imperativen Welt.
Sie besteht aus

- Einem Paar von Pointern, die auf den Anfang und das Ende zeigen
- Aus Elementen, welche bestehen aus
  - Einem Pointer auf das nächste Element (null, falls nicht da)
  - Einem Pointer auf das vorherige Element (null, falls nicht da)

Eine Double-Linked-List ist die klassische Einstiegs-Datenstruktur in der imperativen Welt.
Sie besteht aus

- Einem Paar von Pointern, die auf den Anfang und das Ende zeigen
- Aus Elementen, welche bestehen aus
  - Einem Pointer auf das nächste Element (null, falls nicht da)
  - Einem Pointer auf das vorherige Element (null, falls nicht da)
  - Einem Datum, welches gespeichert werden soll

Grafisch:


NULL
Inhalt
Next
Prev
Inhalt
Next
Prev
Start
Ende
Inhalt
Next
Prev
NULL

Die Vorteile dieser Datenstruktur liegen auf der Hand:

Die Vorteile dieser Datenstruktur liegen auf der Hand:

- Einfügen eines Elementes vorne/hinten ist $\mathcal{O}(1)$

Die Vorteile dieser Datenstruktur liegen auf der Hand:

- Einfügen eines Elementes vorne/hinten ist $\mathcal{O}(1)$
- Iteration (z.B. map) ist einfach

Die Vorteile dieser Datenstruktur liegen auf der Hand:

- Einfügen eines Elementes vorne/hinten ist $\mathcal{O}(1)$
- Iteration (z.B. map) ist einfach
- Finden/Updaten eines Elementes ist $\mathcal{O}(n)$

Die Vorteile dieser Datenstruktur liegen auf der Hand:

- Einfügen eines Elementes vorne/hinten ist $\mathcal{O}(1)$
- Iteration (z.B. map) ist einfach
- Finden/Updaten eines Elementes ist $\mathcal{O}(n)$
- Verbinden von 2 Listen ist $\mathcal{O}(1)$

Die Vorteile dieser Datenstruktur liegen auf der Hand:

- Einfügen eines Elementes vorne/hinten ist $\mathcal{O}(1)$
- Iteration (z.B. map) ist einfach
- Finden/Updaten eines Elementes ist $\mathcal{O}(n)$
- Verbinden von 2 Listen ist $\mathcal{O}(1)$

Die Einfache Liste in Haskell hat auch diese Eigenschaften - allerdings nur von Vorne.

Die Vorteile dieser Datenstruktur liegen auf der Hand:

- Einfügen eines Elementes vorne/hinten ist $\mathcal{O}(1)$
- Iteration (z.B. map) ist einfach
- Finden/Updaten eines Elementes ist $\mathcal{O}(n)$
- Verbinden von 2 Listen ist $\mathcal{O}(1)$

Die Einfache Liste in Haskell hat auch diese Eigenschaften - allerdings nur von Vorne.
Einfügen von hinten und Verkettung läuft immernoch in $\mathcal{O}(n)$.

Die Vorteile dieser Datenstruktur liegen auf der Hand:

- Einfügen eines Elementes vorne/hinten ist $\mathcal{O}(1)$
- Iteration (z.B. map) ist einfach
- Finden/Updaten eines Elementes ist $\mathcal{O}(n)$
- Verbinden von 2 Listen ist $\mathcal{O}(1)$

Die Einfache Liste in Haskell hat auch diese Eigenschaften - allerdings nur von Vorne.
Einfügen von hinten und Verkettung läuft immernoch in $\mathcal{O}(n)$.

Wie bekommen wir nun alle Vorteile nach Haskell-Land? Und wieso gibt es da nichts in der Standard-Library?

Also implementieren wir einfach mal diese Datenstruktur:

Also implementieren wir einfach mal diese Datenstruktur:
Einen Pointer in Haskell bekommen und bearbeiten wir mittels

```
newIORef   :: a -> IO (IORef a)
readIORef  :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()
```

Also implementieren wir einfach mal diese Datenstruktur:
Einen Pointer in Haskell bekommen und bearbeiten wir mittels

```
newIORef    :: a -> IO (IORef a)
readIORef   :: IORef a -> IO a
writeIORef  :: IORef a -> a -> IO ()
```

Ein Doubly-Linked-List ist dann

```
data Entry a = Entry { prev :: IORef (Maybe (Entry a))
                     , data :: a
                     , next :: IORef (Maybe (Entry a))
                     }
data Dll a   = Dll   { first :: IORef (Maybe (Entry a))
                     , last  :: IORef (Maybe (Entry a))
                     }
```

Also implementieren wir einfach mal diese Datenstruktur:
Einen Pointer in Haskell bekommen und bearbeiten wir mittels

```
newIORef    :: a -> IO (IORef a)
readIORef   :: IORef a -> IO a
writeIORef  :: IORef a -> a -> IO ()
```

Ein Doubly-Linked-List ist dann

```
data Entry a = Entry { prev :: IORef (Maybe (Entry a))
                     , data :: a
                     , next :: IORef (Maybe (Entry a))
                     }
data Dll a   = Dll   { first :: IORef (Maybe (Entry a))
                     , last  :: IORef (Maybe (Entry a))
                     }
```

wobei das Maybe-Konstrukt im `Entry` einmal den Pointer zurück und einmal den Pointer nach vorn kennzeichnet und das Maybe-Konstrukt im `Dll` eine leere Liste ermöglicht.

Wir können dies nun runterimplementieren und erhalten ein Interface ähnlich zu

```
newDLL      :: IO (Dll a)
insertFront :: a -> Dll a -> IO (Dll a)
insertBack  :: a -> Dll a -> IO (Dll a)
getFront    :: Dll a -> IO (Maybe a)
getBack     :: Dll a -> IO (Maybe a)
map         :: (a -> b) -> Dll a -> IO (Dll b)
...
```

kreieren.

Wir können dies nun runterimplementieren und erhalten ein Interface ähnlich zu

```
newDLL      :: IO (Dll a)
insertFront :: a -> Dll a -> IO (Dll a)
insertBack  :: a -> Dll a -> IO (Dll a)
getFront    :: Dll a -> IO (Maybe a)
getBack     :: Dll a -> IO (Maybe a)
map         :: (a -> b) -> Dll a -> IO (Dll b)
...
```

kreieren.

Probleme:

- Alles ist in IO (wegen IORefs)

Wir können dies nun runterimplementieren und erhalten ein Interface ähnlich zu

```
newDLL      :: IO (Dll a)
insertFront :: a -> Dll a -> IO (Dll a)
insertBack  :: a -> Dll a -> IO (Dll a)
getFront    :: Dll a -> IO (Maybe a)
getBack     :: Dll a -> IO (Maybe a)
map         :: (a -> b) -> Dll a -> IO (Dll b)
...
```

kreieren.

Probleme:

- Alles ist in IO (wegen IORefs)
- Wir können es aufgrund von IO in keinem puren Code benutzen

Wir können dies nun runterimplementieren und erhalten ein Interface ähnlich zu

```
newDLL      :: IO (Dll a)
insertFront :: a -> Dll a -> IO (Dll a)
insertBack  :: a -> Dll a -> IO (Dll a)
getFront    :: Dll a -> IO (Maybe a)
getBack     :: Dll a -> IO (Maybe a)
map         :: (a -> b) -> Dll a -> IO (Dll b)
...
```

kreieren.

Probleme:

- Alles ist in IO (wegen IORefs)
- Wir können es aufgrund von IO in keinem puren Code benutzen
- Wer garantiert uns, dass die Struktur nur Daten hält und (ggf. nach einem „Patch“) nicht per IO Raketen abschiesst?

Wir können dies nun runterimplementieren und erhalten ein Interface ähnlich zu

```
newDLL      :: IO (Dll a)
insertFront :: a -> Dll a -> IO (Dll a)
insertBack  :: a -> Dll a -> IO (Dll a)
getFront    :: Dll a -> IO (Maybe a)
getBack     :: Dll a -> IO (Maybe a)
map         :: (a -> b) -> Dll a -> IO (Dll b)
...
```

kreieren.

Probleme:

- Alles ist in IO (wegen IORefs)
- Wir können es aufgrund von IO in keinem puren Code benutzen
- Wer garantiert uns, dass die Struktur nur Daten hält und (ggf. nach einem „Patch“) nicht per IO Raketen abschiesst?

Das ist keine gute Lösung! Vor allem nicht Funktional!

Nochmal Vor/Nachteile:

Nochmal Vor/Nachteile:
Vorteile:

- Von den Zugriffszeiten genau das, was wir wollten

Nochmal Vor/Nachteile:
Vorteile:

- Von den Zugriffszeiten genau das, was wir wollten

Nachteile:

- praktisch Unbrauchbar durch IO

Nochmal Vor/Nachteile:
Vorteile:

- Von den Zugriffszeiten genau das, was wir wollten

Nachteile:

- praktisch Unbrauchbar durch IO
- VIEL Speicheraufwändiger als die C-Lösung, durch

Nochmal Vor/Nachteile:
Vorteile:

- Von den Zugriffszeiten genau das, was wir wollten

Nachteile:

- praktisch Unbrauchbar durch IO
- VIEL Speicheraufwändiger als die C-Lösung, durch
  - Zusätzliche Pointer im Maybe

Nochmal Vor/Nachteile:
Vorteile:

- Von den Zugriffszeiten genau das, was wir wollten

Nachteile:

- praktisch Unbrauchbar durch IO
- VIEL Speicheraufwändiger als die C-Lösung, durch
  - Zusätzliche Pointer im Maybe
  - Zusätzliche Pointer durch das IORef

Nochmal Vor/Nachteile:
Vorteile:

- Von den Zugriffszeiten genau das, was wir wollten

Nachteile:

- praktisch Unbrauchbar durch IO
- VIEL Speicheraufwändiger als die C-Lösung, durch
  - Zusätzliche Pointer im Maybe
  - Zusätzliche Pointer durch das IORef
  - statt 3 Pointer (Wert, prev, next) speichern wir 7 (1x prev, 1x next, 2xMaybe, 2xIORef, 1x Wert)

Nochmal Vor/Nachteile:
Vorteile:

- Von den Zugriffszeiten genau das, was wir wollten

Nachteile:

- praktisch Unbrauchbar durch IO
- VIEL Speicheraufwändiger als die C-Lösung, durch
  - Zusätzliche Pointer im Maybe
  - Zusätzliche Pointer durch das IORef
  - statt 3 Pointer (Wert, prev, next) speichern wir 7 (1x prev, 1x next, 2xMaybe, 2xIORef, 1x Wert)
  - „Pointer jagen“ im Speicher kostet zusätzliche Zeit beim Lookup.

Nochmal Vor/Nachteile:
Vorteile:

- Von den Zugriffszeiten genau das, was wir wollten

Nachteile:

- praktisch Unbrauchbar durch IO
- VIEL Speicheraufwändiger als die C-Lösung, durch
  - Zusätzliche Pointer im Maybe
  - Zusätzliche Pointer durch das IORef
  - statt 3 Pointer (Wert, prev, next) speichern wir 7 (1x prev, 1x next, 2xMaybe, 2xIORef, 1x Wert)
  - „Pointer jagen“ im Speicher kostet zusätzliche Zeit beim Lookup.

Insgesamt ist es keine gute Idee die Datenstrukturen aus der imperativen Programmierung einfach nachzubauen.

Nochmal Vor/Nachteile:
Vorteile:

- Von den Zugriffszeiten genau das, was wir wollten

Nachteile:

- praktisch Unbrauchbar durch IO
- VIEL Speicheraufwändiger als die C-Lösung, durch
  - Zusätzliche Pointer im Maybe
  - Zusätzliche Pointer durch das IORef
  - statt 3 Pointer (Wert, prev, next) speichern wir 7 (1x prev, 1x next, 2xMaybe, 2xIORef, 1x Wert)
  - „Pointer jagen“ im Speicher kostet zusätzliche Zeit beim Lookup.

Insgesamt ist es keine gute Idee die Datenstrukturen aus der imperativen Programmierung einfach nachzubauen.
Aber wie geht es dann?

Wir fangen einfach mal an mit einem Wunschkonzert. Wir hätten gerne:

Wir fangen einfach mal an mit einem Wunschkonzert. Wir hätten gerne:

- Ein Sequence-Ähnliches Ding (Array, Liste, etc.)

Wir fangen einfach mal an mit einem Wunschkonzert. Wir hätten gerne:

- Ein Sequence-Ähnliches Ding (Array, Liste, etc.)
- Schnelles Hinzufügen/Entfernen von Elementen am Anfang/Ende

Wir fangen einfach mal an mit einem Wunschkonzert. Wir hätten gerne:

- Ein Sequence-Ähnliches Ding (Array, Liste, etc.)
- Schnelles Hinzufügen/Entfernen von Elementen am Anfang/Ende
- Schnelles Zusammenfügen zweier dieser Dinge

Wir fangen einfach mal an mit einem Wunschkonzert. Wir hätten gerne:

- Ein Sequence-Ähnliches Ding (Array, Liste, etc.)
- Schnelles Hinzufügen/Entfernen von Elementen am Anfang/Ende
- Schnelles Zusammenfügen zweier dieser Dinge
- Iterieren (=map) über dieses Ding

Wir fangen einfach mal an mit einem Wunschkonzert. Wir hätten gerne:

- Ein Sequence-Ähnliches Ding (Array, Liste, etc.)
- Schnelles Hinzufügen/Entfernen von Elementen am Anfang/Ende
- Schnelles Zusammenfügen zweier dieser Dinge
- Iterieren (=map) über dieses Ding
- Möglichkeit Elemente aus der Mitte zu entfernen

Wir fangen einfach mal an mit einem Wunschkonzert. Wir hätten gerne:

- Ein Sequence-Ähnliches Ding (Array, Liste, etc.)
- Schnelles Hinzufügen/Entfernen von Elementen am Anfang/Ende
- Schnelles Zusammenfügen zweier dieser Dinge
- Iterieren (=map) über dieses Ding
- Möglichkeit Elemente aus der Mitte zu entfernen

und weil wir schonmal dabei sind:

- Immutable und pure

Wir fangen einfach mal an mit einem Wunschkonzert. Wir hätten gerne:

- Ein Sequence-Ähnliches Ding (Array, Liste, etc.)
- Schnelles Hinzufügen/Entfernen von Elementen am Anfang/Ende
- Schnelles Zusammenfügen zweier dieser Dinge
- Iterieren (=map) über dieses Ding
- Möglichkeit Elemente aus der Mitte zu entfernen

und weil wir schonmal dabei sind:

- Immutable und pure
- Wenig Speicherverbrauch

Wir fangen einfach mal an mit einem Wunschkonzert. Wir hätten gerne:

- Ein Sequence-Ähnliches Ding (Array, Liste, etc.)
- Schnelles Hinzufügen/Entfernen von Elementen am Anfang/Ende
- Schnelles Zusammenfügen zweier dieser Dinge
- Iterieren (=map) über dieses Ding
- Möglichkeit Elemente aus der Mitte zu entfernen

und weil wir schonmal dabei sind:

- Immutable und pure
- Wenig Speicherverbrauch

Die erste Liste ist die typische Problemstellung deren Lösung eine Doubly-Linked-List ist - mit den Zusatzanforderungen müssen wir uns was anderes überlegen.

Wie könnte so eine API (minimalst) aussehen?

Wie könnte so eine API (minimalst) aussehen?

```
data Ding a = ...

empty     :: Ding a
isEmpty   :: Ding a -> Bool
append    :: Ding a -> a -> Ding a
prepend   :: Ding a -> a -> Ding a
getFirst  :: Ding a -> Maybe (a, Ding a)
getLast   :: Ding a -> Maybe (a, Ding a)
concat    :: Ding a -> Ding a -> Ding a
```

Wenn das alles ist, dann können wir das auch zu einer Typklasse machen

Wenn das alles ist, dann können wir das auch zu einer Typklasse machen

```
class Deque d where
  empty     :: d a
  isEmpty   :: d a -> Bool
  append    :: d a -> a -> d a
  prepend   :: d a -> a -> d a
  getFirst  :: d a -> Maybe (a, d a)
  getLast   :: d a -> Maybe (a, d a)
  concat    :: d a -> d a -> d a
```

Lösung 1: 2 Listen

```
data TwoLists a = TwoLists { front :: [a]
                           , back  :: [a]
                           }
```

Lösung 1: 2 Listen

```
data TwoLists a = TwoLists { front :: [a]
                           , back  :: [a]
                           }
```

Wie implementieren wir jetzt die API?

```
empty = TwoLists [] []
```

```
empty = TwoLists [] []

isEmpty (TwoLists [] []) = True
isEmpty _                = False
```

```haskell
empty = TwoLists [] []

isEmpty (TwoLists [] []) = True
isEmpty _                = False

prepend a (TwoLists front back) = TwoLists (a:front) back
```

```haskell
empty = TwoLists [] []

isEmpty (TwoLists [] []) = True
isEmpty _                = False

prepend a (TwoLists front back) = TwoLists (a:front) back

append  a (TwoLists front back) = TwoLists front (a:back)
```

```
getFirst   (TwoLists []     []) = Nothing
getFirst   (TwoLists (a:as) bs) = Just (a, TwoLists as bs)
getFirst   (TwoLists []     bs  = let (c:cs) = reverse bs in
                                  Just (c, TwoLists cs [])
```

```
getFirst (TwoLists []     []) = Nothing
getFirst (TwoLists (a:as) bs) = Just (a, TwoLists as bs)
getFirst (TwoLists []     bs  = let (c:cs) = reverse bs in
                                Just (c, TwoLists cs [])

getLast  (TwoLists []     []) = Nothing
getLast  (TwoLists as (b:bs)) = Just (b, TwoLists as bs)
getLast  (TwoLists as     []) = let (c:cs) = reverse as in
                                Just (c, TwoLists [] cs)
```

```
getFirst   (TwoLists []     []) = Nothing
getFirst   (TwoLists (a:as) bs) = Just (a, TwoLists as bs)
getFirst   (TwoLists []     bs  = let (c:cs) = reverse bs in
                                  Just (c, TwoLists cs [])

getLast    (TwoLists []     []) = Nothing
getLast    (TwoLists as (b:bs)) = Just (b, TwoLists as bs)
getLast    (TwoLists as     []) = let (c:cs) = reverse as in
                                  Just (c, TwoLists [] cs)

concat (TwoLists as bs) (TwoLists cs ds) =
       TwoLists (as ++ reverse bs) (ds ++ reverse cs)
```

Diese Struktur erfüllt alle Dinge, die wir aus dem Interface haben wollten, aber...

Diese Struktur erfüllt alle Dinge, die wir aus dem Interface haben wollten, aber...

- Wir benutzen `reverse`, welchen Laufzeit $\mathcal{O}(n)$ hat.

Diese Struktur erfüllt alle Dinge, die wir aus dem Interface haben wollten, aber...

- Wir benutzen `reverse`, welchen Laufzeit $\mathcal{O}(n)$ hat.
  Was alle Operationen, die dies benutzen auf $\mathcal{O}(n)$ hochzieht.

Diese Struktur erfüllt alle Dinge, die wir aus dem Interface haben wollten, aber...

- Wir benutzen `reverse`, welchen Laufzeit $\mathcal{O}(n)$ hat.
  Was alle Operationen, die dies benutzen auf $\mathcal{O}(n)$ hochzieht.

- empty $\mathcal{O}(1)$
- append $\mathcal{O}(1)$
- prepend $\mathcal{O}(1)$
- getFirst $\mathcal{O}(n)$
- getLast $\mathcal{O}(n)$
- iterate $\mathcal{O}(n^2)$

Diese Struktur erfüllt alle Dinge, die wir aus dem Interface haben wollten, aber...

- Wir benutzen `reverse`, welchen Laufzeit $\mathcal{O}(n)$ hat.
  Was alle Operationen, die dies benutzen auf $\mathcal{O}(n)$ hochzieht.

- empty $\mathcal{O}(1)$
- append $\mathcal{O}(1)$
- prepend $\mathcal{O}(1)$
- getFirst $\mathcal{O}(n)$
- getLast $\mathcal{O}(n)$
- iterate $\mathcal{O}(n^2)$

Die $\mathcal{O}(n)$-Operation kommt zwar nur selten vor, aber wenn wir z.B. abwechselnd `getFirst` und `getLast` machen, machen wir diese teure Operation jedes mal.

Gibt es da nichts besseres?

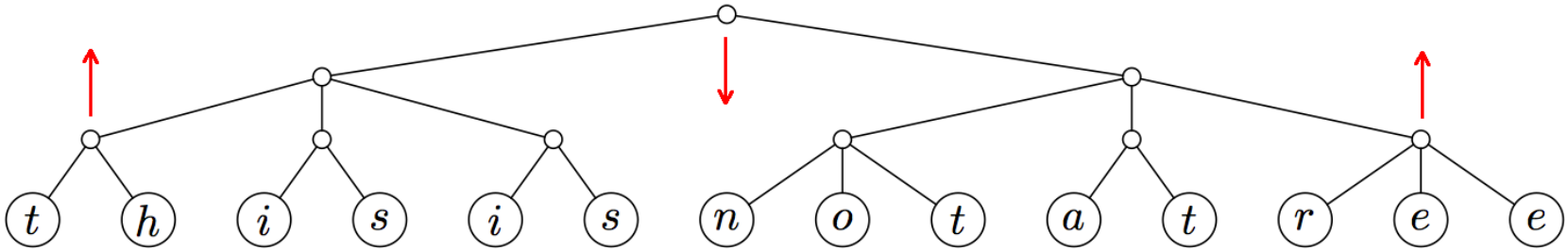
Gibt es da nichts besseres?

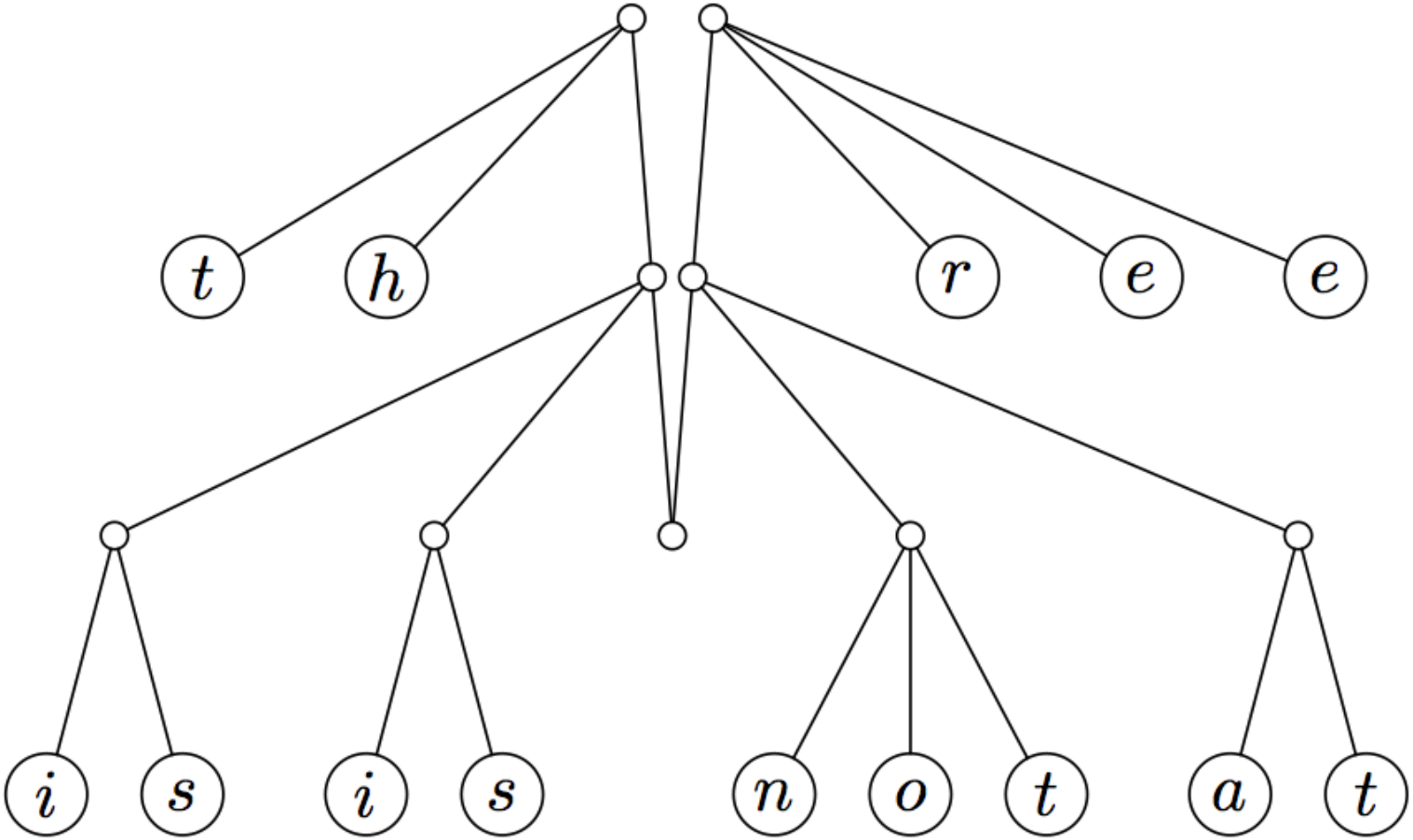
Ja, aber das ist nicht ganz so simpel.

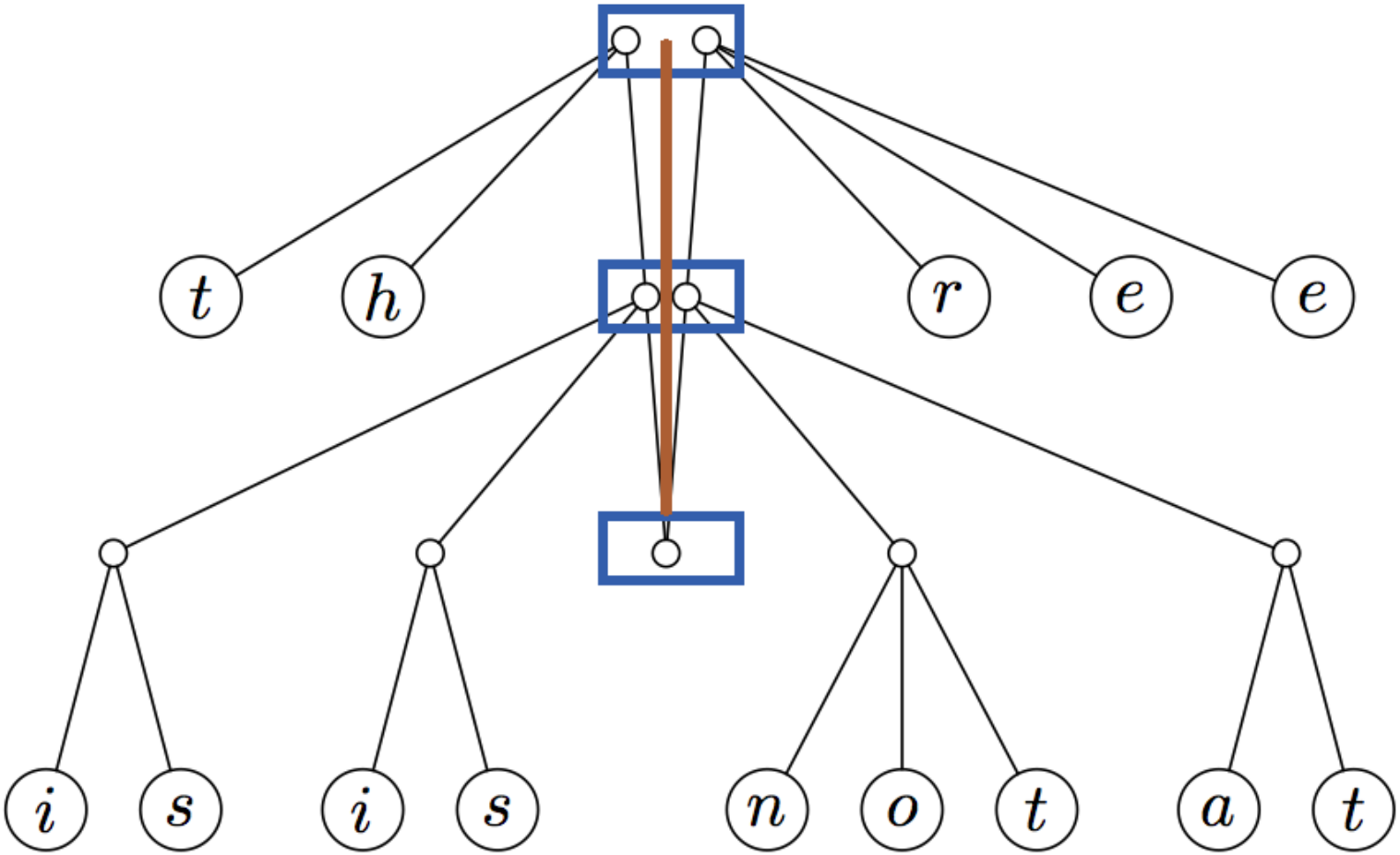
Gibt es da nichts besseres?

Ja, aber das ist nicht ganz so simpel.
Wir müssen dazu einen Exkurs in Bäume machen.

Wir starten hierzu mit einem klassischen 2-3-Tree, also einem Baum mit 2 oder 3 Kind-Knoten in jeder Ebene


t
h
i
s
i
s
n
o
t
a
t
r
e
e


t
h
r
e
e
i
s
i
s
n
o
t
a
t


t
h
r
e
e
i
s
i
s
n
o
t
a
t

So ein Baum hat in jeder Ebene ein Präfix, den Rest des Baumes und ein Suffix.

So ein Baum hat in jeder Ebene ein Präfix, den Rest des Baumes und ein Suffix.
In der ersten Ebene hat er 2 oder 3 Kind-Knoten
In der zweiten Ebene 1 oder 2 Kind-Bäume der Tiefe 1
In der dritten Ebene 1 oder 2 Kind-Bäume der Tiefe 2 etc.

So ein Baum hat in jeder Ebene ein Präfix, den Rest des Baumes und ein Suffix.
In der ersten Ebene hat er 2 oder 3 Kind-Knoten
In der zweiten Ebene 1 oder 2 Kind-Bäume der Tiefe 1
In der dritten Ebene 1 oder 2 Kind-Bäume der Tiefe 2 etc.
Wir nennen Prefixe und Suffixe nun Affixe und sagen, dass diese bis zu 4 Elemente haben können. Dies bringt uns später Vorteile bei der Laufzeit.

Kurz in Code, was wir bisher definiert haben:

```
data Node  a = Branch2 a a
             | Branch3 a a a
           deriving Show
```

Kurz in Code, was wir bisher definiert haben:

```
data Node  a = Branch2 a a
             | Branch3 a a a
             deriving Show

data Affix a = One   a
             | Two   a a
             | Three a a a
             | Four  a a a a
             deriving Show
```

Kurz in Code, was wir bisher definiert haben:

```
data Node  a = Branch2 a a
             | Branch3 a a a
             deriving Show

data Affix a = One   a
             | Two   a a
             | Three a a a
             | Four  a a a a
             deriving Show

data FingerTree a
  = Empty        -- We can have empty trees.
  | Single a     -- We need a special case for trees of size one.
  -- The common case with a prefix, suffix, and a deeper tree.
  | Deep {
    prefix :: Affix a,                 -- Values on the left.
    deeper :: FingerTree (Node a),     -- The deeper finger tree,
                                       -- storing deeper 2-3 trees.
    suffix :: Affix a                  -- Values on the right.
  }
  deriving Show
```

Was haben wir hiermit?

Was haben wir hiermit?

- Einen Baum, der uns die „Enden“ direkt präsentiert

Was haben wir hiermit?

- Einen Baum, der uns die „Enden“ direkt präsentiert
- Jede Ebene `Deep` fügt eine weitere `Node` hinzu

Was haben wir hiermit?

- Einen Baum, der uns die „Enden“ direkt präsentiert
- Jede Ebene `Deep` fügt eine weitere `Node` hinzu
  `FingerTree (Node a)` hat einen Affix von $2 \cdot 1$ (`One (Branch2 a)`) bis $3 \cdot 4$ ((`Four (Branch3 a)`) Elementen

Was haben wir hiermit?

- Einen Baum, der uns die „Enden“ direkt präsentiert
- Jede Ebene `Deep` fügt eine weitere `Node` hinzu
  `FingerTree (Node a)` hat einen Affix von $2 \cdot 1$ (`One (Branch2 a)`) bis $3 \cdot 4$ ((`Four (Branch3 a)`) Elementen
  `FingerTree (Node (Node a))` hat einen Affix von $2 \cdot 2 \cdot 1$ bis $3 \cdot 3 \cdot 4$ Elementen

Was haben wir hiermit?

- Einen Baum, der uns die „Enden“ direkt präsentiert
- Jede Ebene `Deep` fügt eine weitere `Node` hinzu
  `FingerTree (Node a)` hat einen Affix von $2 \cdot 1$ (`One (Branch2 a)`) bis $3 \cdot 4$ ((`Four (Branch3 a)`) Elementen
  `FingerTree (Node (Node a))` hat einen Affix von $2 \cdot 2 \cdot 1$ bis $3 \cdot 3 \cdot 4$ Elementen
  etc.

Was haben wir hiermit?

- Einen Baum, der uns die „Enden“ direkt präsentiert
- Jede Ebene `Deep` fügt eine weitere `Node` hinzu
  `FingerTree (Node a)` hat einen Affix von $2 \cdot 1$ (`One (Branch2 a)`) bis $3 \cdot 4$ (`(Four (Branch3 a)`) Elementen
  `FingerTree (Node (Node a))` hat einen Affix von $2 \cdot 2 \cdot 1$ bis $3 \cdot 3 \cdot 4$ Elementen
  etc.
- Wir haben auf jeder Ebene bis zu 3x mehr Elemente als in den vorhergehenden.

Was haben wir hiermit?

- Einen Baum, der uns die „Enden“ direkt präsentiert
- Jede Ebene `Deep` fügt eine weitere `Node` hinzu
  `FingerTree (Node a)` hat einen Affix von $2 \cdot 1$ (`One (Branch2 a)`) bis $3 \cdot 4$ (`(Four (Branch3 a)`) Elementen
  `FingerTree (Node (Node a))` hat einen Affix von $2 \cdot 2 \cdot 1$ bis $3 \cdot 3 \cdot 4$ Elementen
  etc.
- Wir haben auf jeder Ebene bis zu 3x mehr Elemente als in den vorhergehenden.
- Der Baum ist automatisch balanciert (nicht perfekt), da jede Ebene sowohl Präfix als auch Suffix-Elemente haben muss

Wie fügen wir nun Elemente ein?

Wie fügen wir nun Elemente ein?

```
infixr 5 <|
(<|) :: a -> FingerTree a -> FingerTree a

x <| Empty = Single x
x <| Single y = Deep (One x) Empty (One y)
x <| Deep (Four a b c d) deeper suffix =
  Deep (Two x a) (node <| deeper) suffix
  where
    node = Branch3 b c d
x <| tree = tree { prefix = affixPrepend x $ prefix tree }
```

gegeben eine Funktion `affixPrepend`, die aus einem `One` ein `Two` macht etc.

Analog hinten

```
infixr 5 |>
(|>) :: FingerTree a -> a -> FingerTree a

Empty |> x = Single x
Single y |> x = Deep (One y) Empty (One x)
Deep prefix deeper (Four a b c d) |> x =
  Deep prefix (deeper |> node) (Two d x)
  where
    node = Branch3 a b c
tree |> x = tree { suffix = affixAppend x $ suffix tree }
```

gegeben eine Funktion `affixAppend`, die aus einem `One` ein `Two` macht etc.

Aber wie schaut denn nun die Laufzeit aus? Dieses einfügen führt doch im schlimmsten Falle dazu, dass alle Ebenen angefasst und neu aufgebaut werden müssen!

Aber wie schaut denn nun die Laufzeit aus? Dieses einfügen führt doch im schlimmsten Falle dazu, dass alle Ebenen angefasst und neu aufgebaut werden müssen!
Richtig. Aber wie häufig?

Aber wie schaut denn nun die Laufzeit aus? Dieses einfügen führt doch im schlimmsten Falle dazu, dass alle Ebenen angefasst und neu aufgebaut werden müssen!
Richtig. Aber wie häufig?
Jede Operation macht eine $\mathcal{O}(1)$-Operation in der obersten Ebene.
Mit Wahrscheinlichkeit $\frac{1}{2}$ machen wir auch in Ebene 2 eine $\mathcal{O}(1)$-Operation.

Generell machen wir für m Einfüge-Operationen (von derselben Seite)

$$T = m + \frac{1}{2}m + \frac{1}{4}m + ... = \sum_{i=0}^{m} \frac{1}{2^i}m$$

Generell machen wir für m Einfüge-Operationen (von derselben Seite)

$$T = m + \frac{1}{2}m + \frac{1}{4}m + ... = \sum_{i=0}^{m} \frac{1}{2^i} m$$

Was uns allen bekannt vorkommen sollte. Für $m \to \infty$ Einfügeoperationen brauchen wir folglich

$$m \cdot \lim_{m \to \infty} \sum_{i=0}^{m} \frac{1}{2^i} = m \cdot 2$$

Operationen, welches uns armortisiert pro Operation $\mathcal{O}(1)$ kostet.

Das letzte bzw. erste Element bekommen ist ähnlich kompliziert und mit derselben Argumentation kommt man auch hier auf eine Laufzeit von $\mathcal{O}(1)$ für beide Operationen.

Das letzte bzw. erste Element bekommen ist ähnlich kompliziert und mit derselben Argumentation kommt man auch hier auf eine Laufzeit von $\mathcal{O}(1)$ für beide Operationen.
Ich werde hier nur kurz den Code zeigen, damit er auf den Folien ist. Weitere Informationen findet man in dem (sehr guten!) Blogpost von Andrew Gibiansky unter
`http://andrew.gibiansky.com/blog/haskell/finger-trees/`

Zunächst definieren wir uns eine View-Datenstruktur:

```
data View a = Nil | View a (FingerTree a)
  deriving Show
```

welches einfach nur ein schöne Variante des von `Maybe (a, FingerTree a)` ist.

```haskell
viewl :: FingerTree a -> View a
viewl Empty = Nil
viewl (Single x) = View x Empty
viewl (Deep (One x) deeper suffix) = View x rest
  where
    rest =
      case viewl deeper of
        View node rest' ->
          Deep (fromList $ toList node) rest' suffix
        Nil -> case suffix of
          (One x) -> Single x
          (Two x y) -> Deep (One x) Empty (One y)
          (Three x y z) -> Deep (Two x y) Empty (One z)
          (Four x y z w) -> Deep (Three x y z) Empty (One w)
viewl (Deep prefix deeper suffix) =
  View first $ Deep (fromList rest) deeper suffix
  where
    first:rest = toList prefix
```

mit fromList und toList definiert auf Affix

Der Code für `viewr` ist analog hierzu.

Der Code für `viewr` ist analog hierzu.
Wir haben noch das verketten von 2 Finger-Trees:
Da wir in $\mathcal{O}(1)$ view und append machen können, können wir in $\mathcal{O}(m)$ Zeit einen Baum der Länge m an einen Baum der Länge n anhängen.

Der Code für `viewr` ist analog hierzu.
Wir haben noch das verketten von 2 Finger-Trees:
Da wir in $\mathcal{O}(1)$ view und append machen können, können wir in $\mathcal{O}(m)$ Zeit einen Baum der Länge m an einen Baum der Länge n anhängen.

Dies geht aber auch geschickter über die Struktur des Trees. Für den Code sei auf die Literatur verwiesen. Hier nur die Laufzeit: $\mathcal{O}(\log(min(n, m)))$.

Mit dieser Struktur geht sogar noch (viel!) mehr.

Mit dieser Struktur geht sogar noch (viel!) mehr.
Gegeben einen Monoid m können wir Annotationen von Typ m hinzufügen.

Mit dieser Struktur geht sogar noch (viel!) mehr.

Gegeben einen Monoid *m* können wir Annotationen von Typ *m* hinzufügen.

Wenn wir dann weiterhin so etwas wie

```
class Monoid v => Measured a v where
  measure :: a -> v
```

vorraussetzen, dann ergibt sich insgesamt

```haskell
data Node v a = Branch3 v a a a
              | Branch2 v a a
              deriving Show

data FingerTree v a
  = Empty
  | Single a
  | Deep {
    annotation :: v, -- Add an annotation to each branch.
    prefix :: Affix a,
    deeper :: FingerTree v (Node v a),
    suffix :: Affix a
  }
  deriving Show
```

für die Definition des FingerTrees.

```haskell
data Node v a = Branch3 v a a a
              | Branch2 v a a
              deriving Show

data FingerTree v a
  = Empty
  | Single a
  | Deep {
    annotation :: v, -- Add an annotation to each branch.
    prefix :: Affix a,
    deeper :: FingerTree v (Node v a),
    suffix :: Affix a
  }
  deriving Show
```

für die Definition des FingerTrees.
Alle bisherigen Operationen müssen natürlich angepasst werden, um die Annotationen mit durchzuschleifen. An der Laufzeit ändert sich nichts.

Einen FingerTree kann man dann messen durch

```
instance Measured a v => Measured (FingerTree v a) v where
  measure Empty = empty
  measure (Single x) = measure x
  measure tree = annotation tree

instance Measured a v => Measured (Node v a) v where
  measure (Branch2 v _ _) = v
  measure (Branch3 v _ _ _) = v
```

Frage: Was bringt uns das ganze?

Frage: Was bringt uns das ganze?
Mittels

```haskell
-- Monoidal size - all leaves have Size 1.
newtype Size = Size Int deriving (Show, Eq, Ord)

-- Storage for our values.
newtype Value a = Value a deriving Show

-- Sizes just add normally.
instance Monoid Size where
  mempty = Size 0
  Size x <> Size y = Size $ x + y

-- All values just have size one.
instance Measured (Value a) Size where
  measure _ = Size 1
```

Nummerieren wir die Elemente durch. Somit können wir in $\mathcal{O}(log(n))$ auf das n-te Element zugreifen!

Frage: Was bringt uns das ganze (2)?

Frage: Was bringt uns das ganze (2)?
Mittels

```
data Prioritized a = Prioritized {
    priority :: Int,
    item :: a }
data Priority = NegativeInfinity | Priority Int deriving Eq
instance Monoid Priority where
  NegativeInfinity <> x = x
  x <> NegativeInfinity = x
  (Priority x) <> (Priority y) = Priority $ max x y
  empty = NegativeInfinity
instance Measured (Prioritized a) Priority where
  measure = Priority . priority
newtype PriorityQueue a = PriorityQueue (FingerTree Priority
                                         (Prioritized a))
```

wandeln wir den FingerTree in eine Priority-Queue, wo wir schnell auf das Element mit der höchsten Priorität zugreifen können.

Frage: Was bringt uns das ganze (3)?

Frage: Was bringt uns das ganze (3)?
Da wir Objekte schnell lokalisieren können (in $\mathcal{O}(\log(n))$), können wir auch einen schnellen split an so einer Stelle machen.

Was haben wir gelernt?

Was haben wir gelernt?
Die simpelste Lösung ist nicht immer die Beste, aber wir haben ausgehend von doppelt-verketteten Listen eine gute Datenstruktur gefunden.

Was haben wir gelernt?
Die simpelste Lösung ist nicht immer die Beste, aber wir haben ausgehend von doppelt-verketteten Listen eine gute Datenstruktur gefunden.
Abschließend ein kurzer Überblick über Laufzeiten gängiger Strukturen.

Operation	Amortized Bounds			
	Finger Tree	Annotated 2-3 Tree	List	Vector
cons/snoc	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)/\mathcal{O}(n)$	$\mathcal{O}(n)$
viewl/viewr	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)/\mathcal{O}(n)$	$\mathcal{O}(1)$
measure/lenth	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
append	$\mathcal{O}(\log \min(l_1, l_2))$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n+m)$
split	$\mathcal{O}(\log \min(n, l-n))$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
fromList/toList/reverse	$\mathcal{O}(l)/\mathcal{O}(l)/\mathcal{O}(l)$	$\mathcal{O}(l)$	$\mathcal{O}(1)/\mathcal{O}(1)/\mathcal{O}(n)$	$\mathcal{O}(n)$
index	$\mathcal{O}(\log \min(n, l-n))$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$

FingerTrees sind die Datenstruktur, wenn man viele Anfüge und Entfern-Operationen hat. Daher basiert auch alles in `Data.Sequence` intern auf FingerTrees.

FingerTrees sind die Datenstruktur, wenn man viele Anfüge und Entfern-Operationen hat. Daher basiert auch alles in `Data.Sequence` intern auf FingerTrees.
Auch sind sie Doppelt-Verketteten Listen in allen praktischen Anwendungsbereichen überlegen, da sie auch effizienten Random-Access erlauben ($\mathcal{O}(\log n)$ statt $\mathcal{O}(n)$) und sich in $\mathcal{O}(\log n)$ zerteilen lassen (statt $\mathcal{O}(n)$).

Kommen wir nun zu einem anderen Thema: Laziness

Kommen wir nun zu einem anderen Thema: Laziness