Tisztán funkcionális adatszerkezetek

Bevezetés

A hatékony adatszerkezetek általában...

„[..] language-independent only in the sense of Henry Ford: Programmers can use any language as they want, as long as it's imperative." – Chris Okasaki, *Purely Functional Data Structures*, 1996

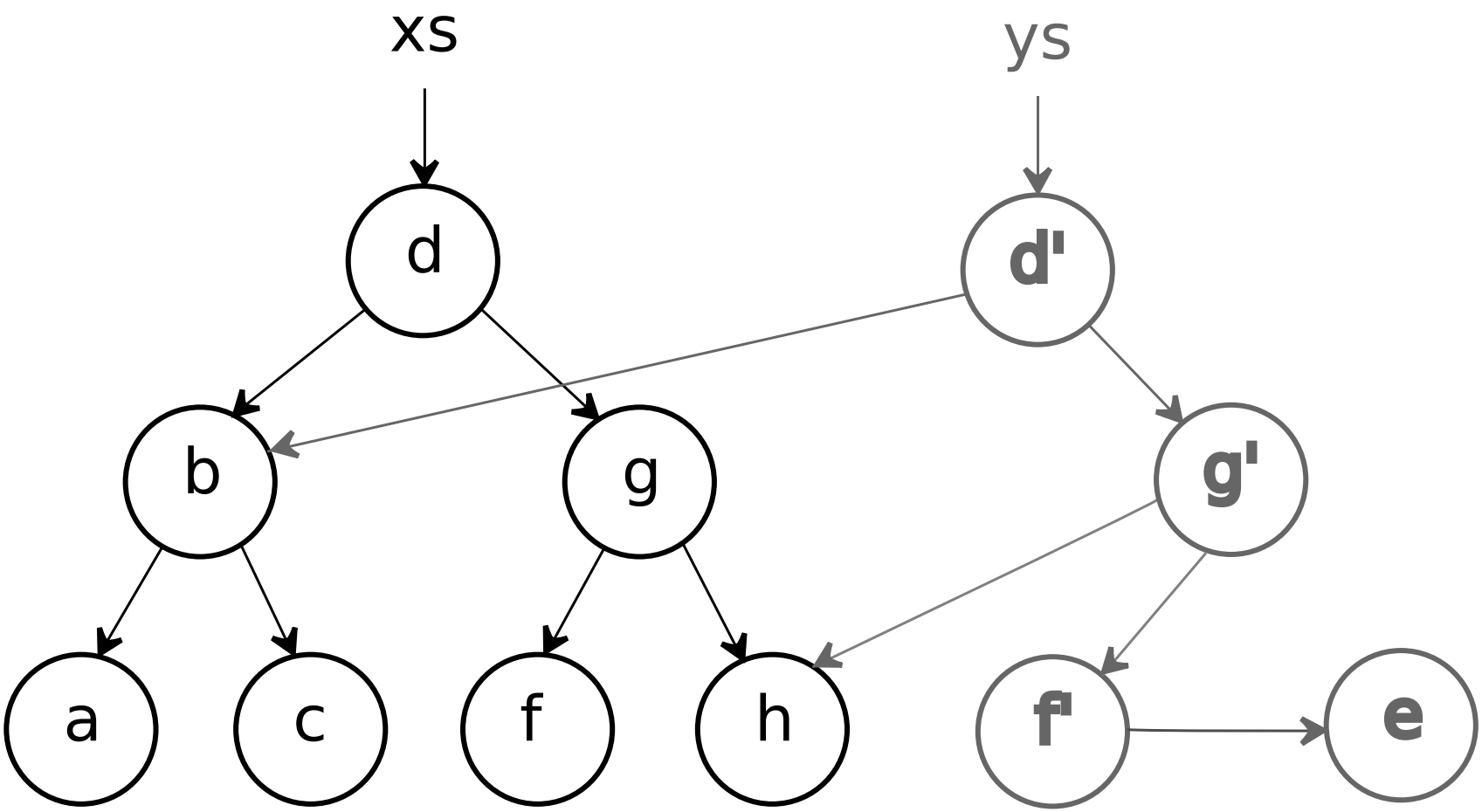
Az imperatív adatszerkezetek viszont:

- Gyakran döntő mértékben támaszkodnak az elemek felülírására („destructive update")
- Ezáltal nem tisztán funkcionálisak, mivel azok mindig (automatikusan) *perzisztens*ek: az adatszerkezet régi állapotai is elérhetőek

Hatékony (megvalósítható) perzisztencia: *lusta kiértékelés*, *memoization*, *amortizáció*.

Hogyan és hol jelenik meg a perzisztencia?

$$xs = fromList\ [a, b, c, d, f, g, h]$$
$$ys = insert\ e\ xs$$



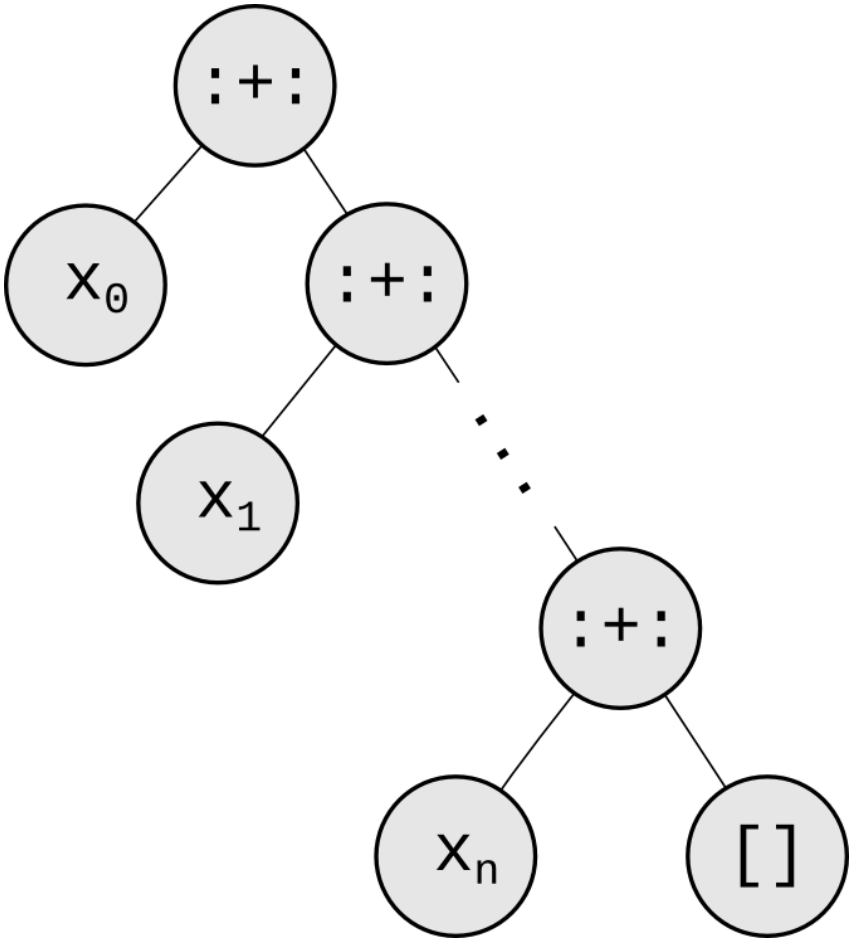
Egy kis ismétlés: listák

$$\textbf{data}\ [\alpha] = \alpha\ :+:\ [\alpha] \mid []$$

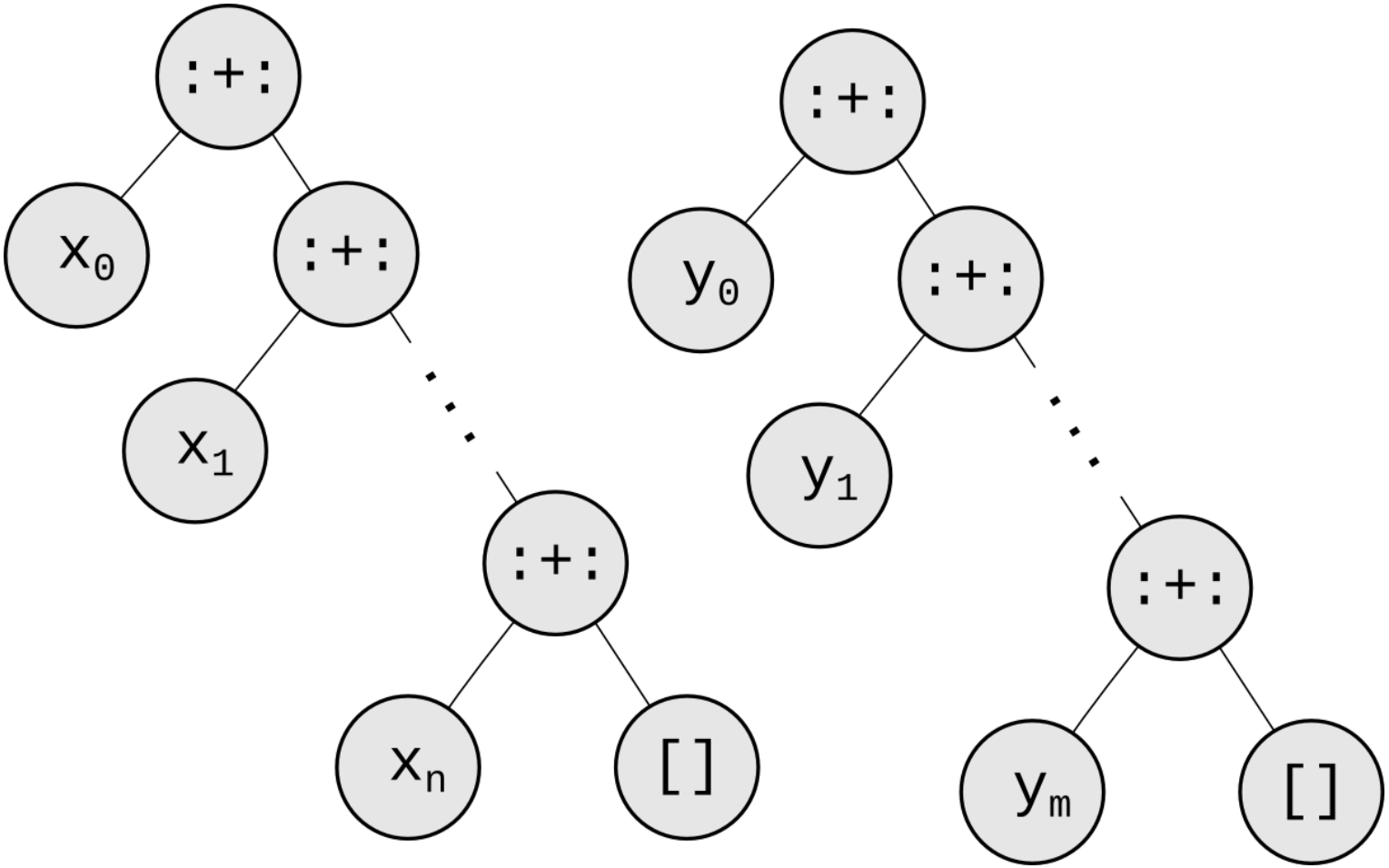
$$(:) :: \alpha \to [\alpha] \to [\alpha]$$
$$x : xs = x\ :+:\ xs$$

$$(+\!\!+) :: [\alpha] \to [\alpha] \to [\alpha]$$
$$[] +\!\!+ ys = ys$$
$$(x\ :+:\ xs) +\!\!+ ys = x : (xs +\!\!+ ys)$$

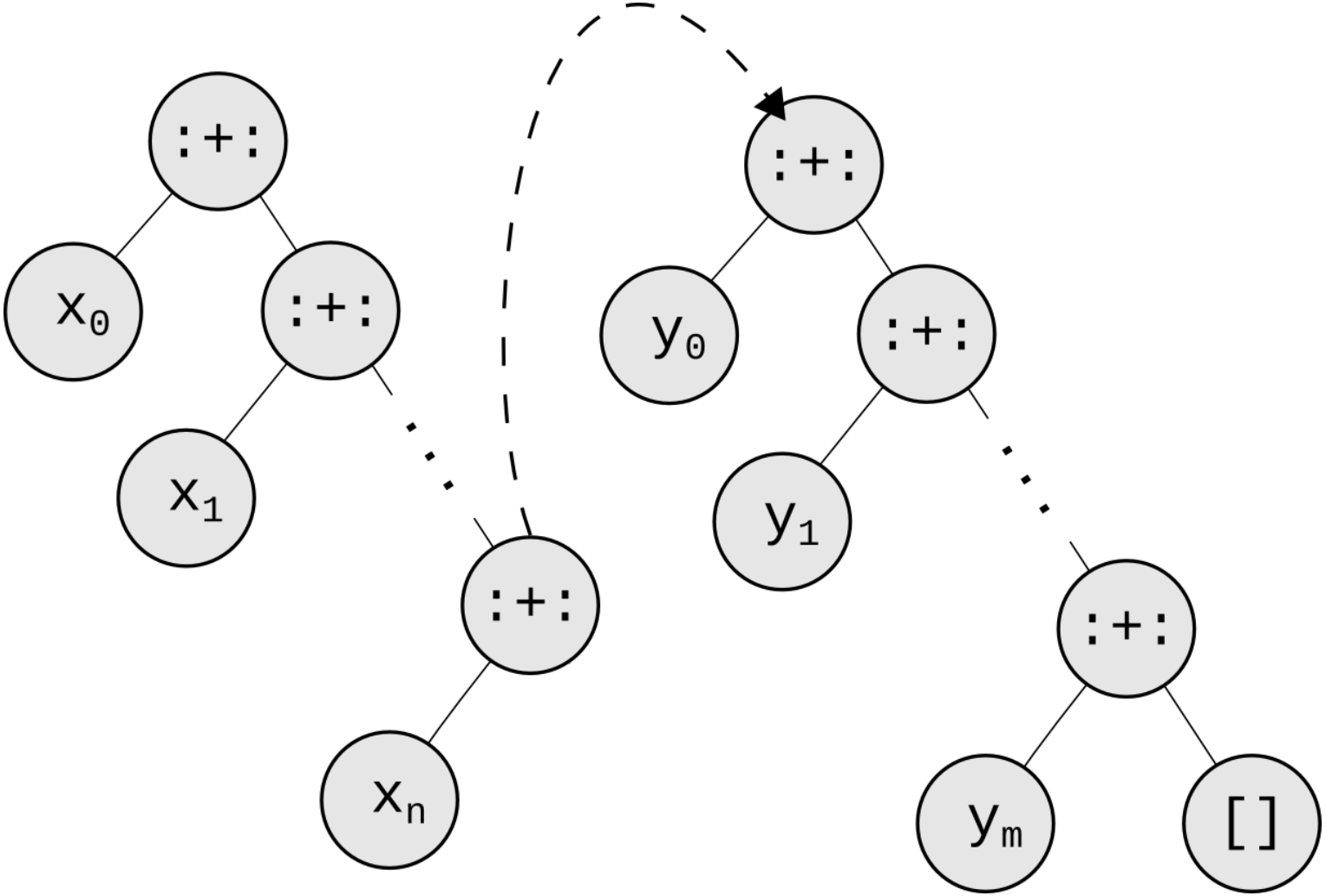
Egy kis ismétlés: listák



Egy kis ismétlés: listák



Egy kis ismétlés: listák



Listák: a költségek elemzése

$xs_1 = xs_0 +\!\!+ ys_1$
-- költség: $length\ xs_0$

$xs_2 = xs_1 +\!\!+ ys_2$
-- költség: $length\ (xs_0 +\!\!+ ys_1)$

$xs_3 = xs_2 +\!\!+ ys_3$
-- költség: $length\ (xs_0 +\!\!+ ys_1 +\!\!+ ys_2)$

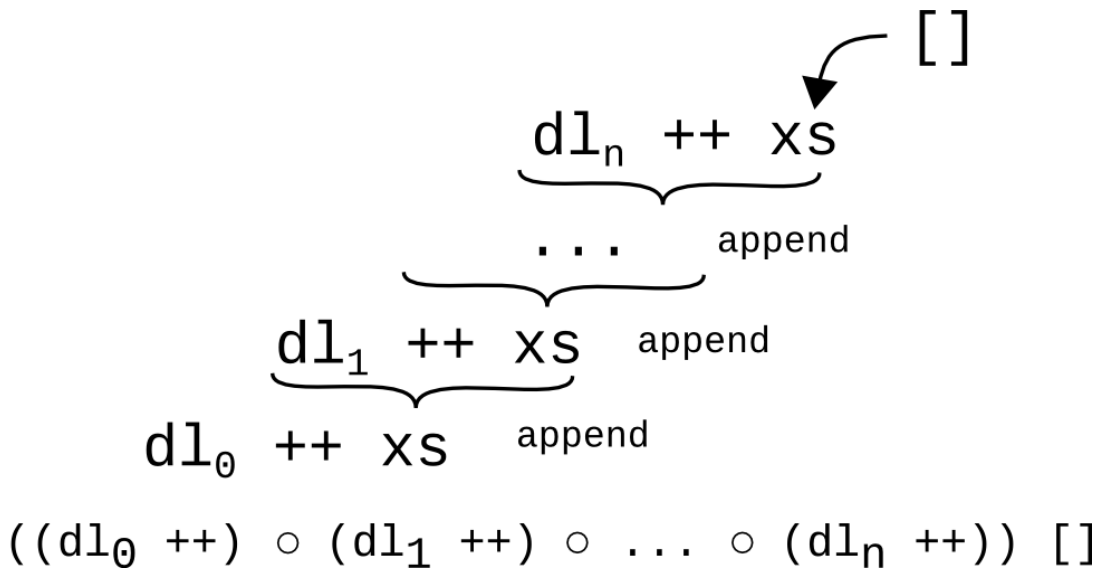
$xs_N = xs_{N-1} +\!\!+ ys_N$
-- költség: $length\ (xs_0 +\!\!+ ys_1 +\!\!+ \ldots +\!\!+ ys_{N-1})$

Továbbfejlesztés: differencialisták

A *differencialista* egy olyan *füuggvény*, amely visszaadja a benne tárolt listát a paramétereként kapott lista után fűzve.

-- http://hackage.haskell.org/package/dlist
newtype *DiffList* α = *DFL* $([\alpha] \rightarrow [\alpha])$

Konstans idejű hozzáfűzést tesz lehetővé a lista elejére és a végére.



Differencialista: a költségek elemzése (áttekintés)

$xs_1 = (fromList\ xs_0)\ `append`\ (fromList\ ys_1)$
-- költség: 1

$xs_2 = (fromList\ xs_1)\ `append`\ (fromList\ ys_2)$
-- költség: 1

$xs_3 = (fromList\ xs_2)\ `append`\ (fromList\ ys_3)$
-- költség: 1

$xs_N = (fromList\ xs_{N-1})\ `append`\ (fromList\ ys_N)$
-- költség: 1

$toList\ xs_N$
-- költség: $length\ (xs_0 +\!\!+ ys_1 +\!\!+ \ldots +\!\!+ ys_N)$

Differencialista: implementáció (1)

$$
\begin{aligned}
&\mathit{fromList} :: [\alpha] \to \mathit{DiffList}\ \alpha \\
&\mathit{fromList}\ \mathit{dl} = \mathit{DFL}\ (\lambda\ \mathit{xs}\ .\ \mathit{dl} \mathbin{+\!\!+} \mathit{xs})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{apply} :: \mathit{DiffList}\ \alpha \to [\alpha] \to [\alpha] \\
&\mathit{apply}\ (\mathit{DFL}\ f)\ \mathit{xs} = f\ \mathit{xs}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{toList} :: \mathit{DiffList}\ \alpha \to [\alpha] \\
&\mathit{toList}\ \mathit{df} = \mathit{apply}\ \mathit{df}\ []
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{empty} :: \mathit{DiffList}\ \alpha \\
&\mathit{empty} = \mathit{DFL}\ (\lambda\ \mathit{xs}\ .\ \mathit{xs})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{singleton} :: \alpha \to \mathit{DiffList}\ \alpha \\
&\mathit{singleton}\ x = \mathit{DFL}\ (\lambda\ \mathit{xs}\ .\ x : \mathit{xs})
\end{aligned}
$$

Differencialista: implementáció (2)

$$
\begin{aligned}
&\textbf{infixr}\ \text{`}\mathit{cons}\text{`} \\
&\mathit{cons} :: \alpha \to \mathit{DiffList}\ \alpha \to \mathit{DiffList}\ \alpha \\
&x\ \text{`}\mathit{cons}\text{`}\ (\mathit{DFL}\ f) = \mathit{DFL}\ ((x\ :) \circ f)
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{infixl}\ \text{`}\mathit{snoc}\text{`} \\
&\mathit{snoc} :: \mathit{DiffList}\ \alpha \to \alpha \to \mathit{DiffList}\ \alpha \\
&(\mathit{DFL}\ f)\ \text{`}\mathit{snoc}\text{`}\ x = \mathit{DFL}\ (f \circ (x\ :))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{append} :: \mathit{DiffList}\ \alpha \to \mathit{DiffList}\ \alpha \to \mathit{DiffList}\ \alpha \\
&\mathit{append}\ (\mathit{DFL}\ f)\ (\mathit{DFL}\ g) = \mathit{DFL}\ (f \circ g)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{concat} :: [\mathit{DiffList}\ \alpha] \to \mathit{DiffList}\ \alpha \\
&\mathit{concat} = \mathit{Data.List.foldr\ append\ empty}
\end{aligned}
$$

Differencialista: implementáció (3)

$$
\begin{aligned}
&\mathit{replicate} :: \mathit{Int} \to \alpha \to \mathit{DiffList}\ \alpha \\
&\mathit{replicate}\ n\ x = \mathit{DFL}\ \$\ \lambda\ \mathit{xs}\ . \\
&\quad \textbf{let}\ f\ m \mid m \leq 0 \quad\ = \mathit{xs} \\
&\qquad\qquad \mid \mathit{otherwise} = x : f\ (m - 1) \\
&\quad \textbf{in}\ f\ n
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{foldr} :: (\alpha \to \beta \to \beta) \to \beta \to \mathit{DiffList}\ \alpha \to \beta \\
&\mathit{foldr}\ f\ b = \mathit{Data.List.foldr}\ f\ b \circ \mathit{toList}
\end{aligned}
$$

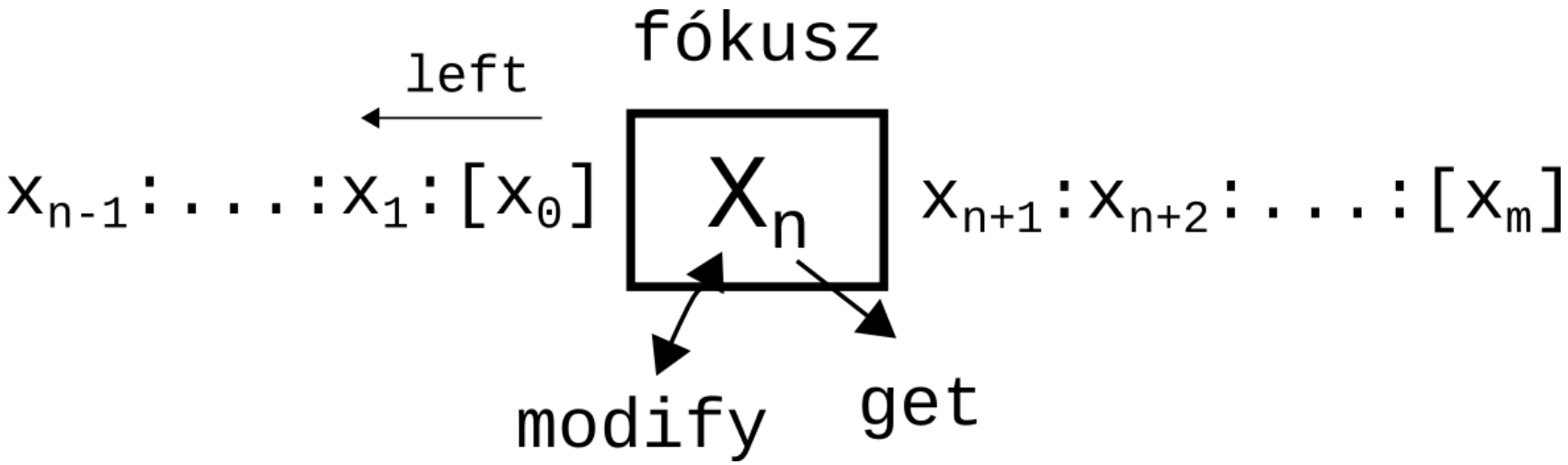
$$
\begin{aligned}
&\mathit{map} :: (\alpha \to \beta) \to \mathit{DiffList}\ \alpha \to \mathit{DiffList}\ \beta \\
&\mathit{map}\ f = \mathit{foldr}\ (\mathit{cons} \circ f)\ \mathit{empty}
\end{aligned}
$$

Interlude: Zipper

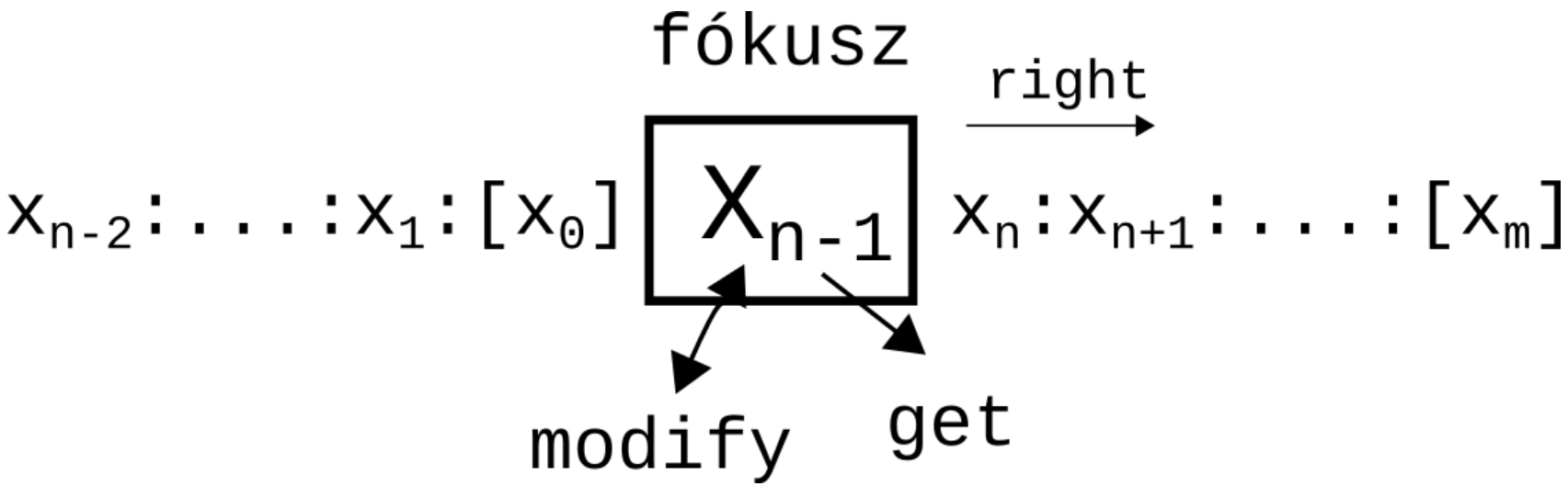
A *zipper* az adatszerkezet egy olyan ábrázolási módja, amely megkönnyíti annak bejárását és elemeinek módosítását.

- "gap buffer": ugyanazon pozícióhoz (kurzorhoz) tartozó beszúrások és törlések nyalábolása, optimalizációja – az amortizált költsége kevés.
- Gyakran alkalmazzák nagyobb méretű adatszerkezetekben való mozgásra vagy „fókuszálásra”, például:
  - Fókusz és ablakok elhelyezésének kezelése (xmonad)
  - Szövegszerkesztők
  - Tranzakciós szemantikával rendelkező állományrendszerek – `http://hackage.haskell.org/package/ZFS`
- Ez a megoldás tetszőleges rekurzívan definiált adatszerkezet (lista, fa stb.) esetén alkalmazható.

Kétirányú listák: "List with Zipper"



Kétirányú listák: "List with Zipper"



Kétirányú listák: implementáció (1)

$$\textbf{newtype}\ \mathit{ZippList}\ \alpha\ =\ \mathit{ZPL}\ ([\alpha],\ [\alpha])$$

$$\mathit{fromList}\ ::\ [\alpha]\ \rightarrow\ \mathit{ZippList}\ \alpha$$
$$\mathit{fromList}\ \mathit{xs}\ =\ \mathit{ZPL}\ ([],\ \mathit{xs})$$

$$\mathit{toList}\ ::\ \mathit{ZippList}\ \alpha\ \rightarrow\ [\alpha]$$
$$\mathit{toList}\ (\mathit{ZPL}\ (\mathit{xs},\ \mathit{ys}))\ =\ \mathit{reverse}\ \mathit{xs}\ +\!\!+\ \mathit{ys}$$

$$\mathit{get}\ ::\ \mathit{ZippList}\ \alpha\ \rightarrow\ [\alpha]$$
$$\mathit{get}\ (\mathit{ZPL}\ (\_,\ \mathit{ys}))\ =\ \mathit{ys}$$

Kétirányú listák: implementáció (2)

$$
\begin{aligned}
&\mathit{right} :: \mathit{ZippList}\ \alpha \to \mathit{ZippList}\ \alpha \\
&\mathit{right}\ (\mathit{ZPL}\ (\mathit{xs},\ (y : \mathit{ys}))) = \mathit{ZPL}\ (y : \mathit{xs},\ \mathit{ys}) \\
&\mathit{right}\ \mathit{zl} = \mathit{zl}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{left} :: \mathit{ZippList}\ \alpha \to \mathit{ZippList}\ \alpha \\
&\mathit{left}\ (\mathit{ZPL}\ ((x : \mathit{xs}),\ \mathit{ys})) = \mathit{ZPL}\ (\mathit{xs},\ x : \mathit{ys}) \\
&\mathit{left}\ \mathit{zl} = \mathit{zl}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{put} :: \mathit{Either}\ \alpha\ \alpha \to \mathit{ZippList}\ \alpha \to \mathit{ZippList}\ \alpha \\
&\mathit{put}\ (\mathit{Left}\ e)\ \ (\mathit{ZPL}\ (\mathit{xs},\ \mathit{ys})) = \mathit{ZPL}\ (e : \mathit{xs},\ \mathit{ys}) \\
&\mathit{put}\ (\mathit{Right}\ e)\ (\mathit{ZPL}\ (\mathit{xs},\ \mathit{ys})) = \mathit{ZPL}\ (\mathit{xs},\ e : \mathit{ys})
\end{aligned}
$$

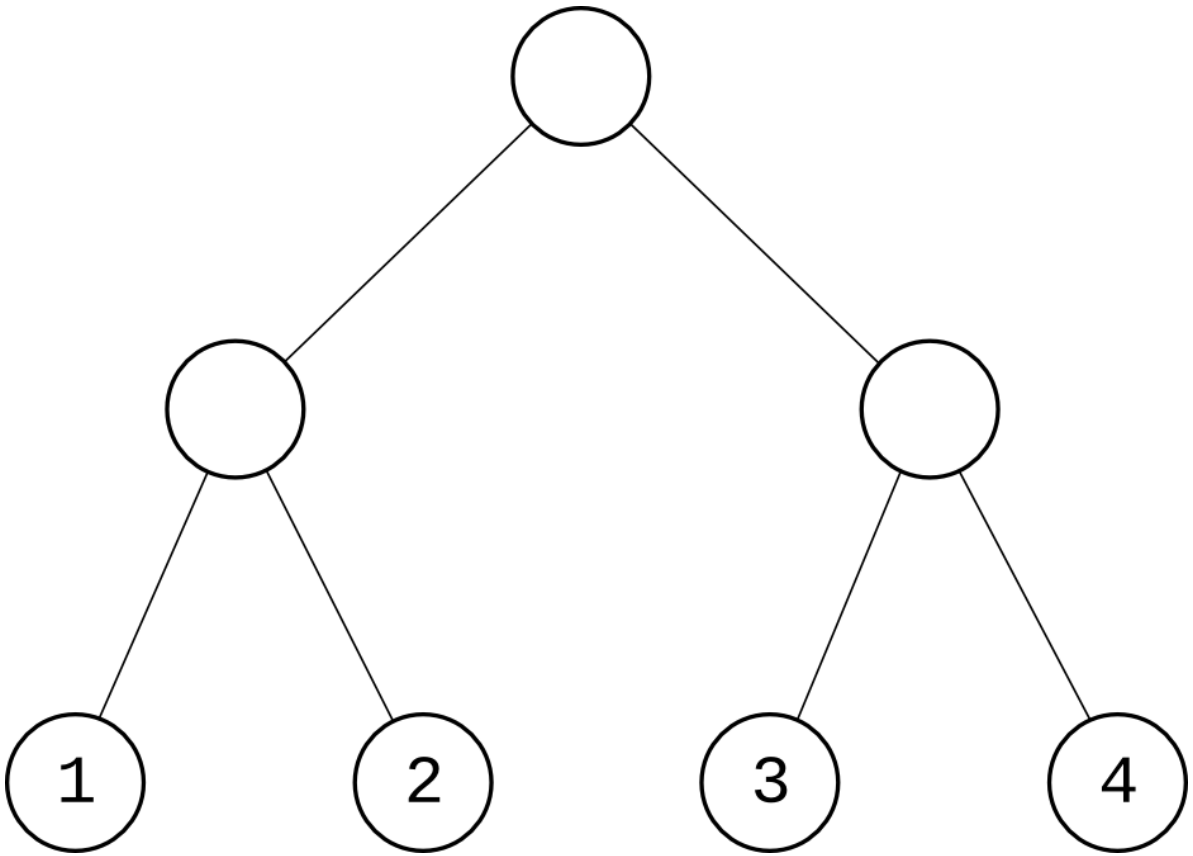
$$
\begin{aligned}
&\mathit{putLeft} = \mathit{put} \circ \mathit{Left} \\
&\mathit{putRight} = \mathit{put} \circ \mathit{Right}
\end{aligned}
$$

Kétirányú listák: implementáció (3)

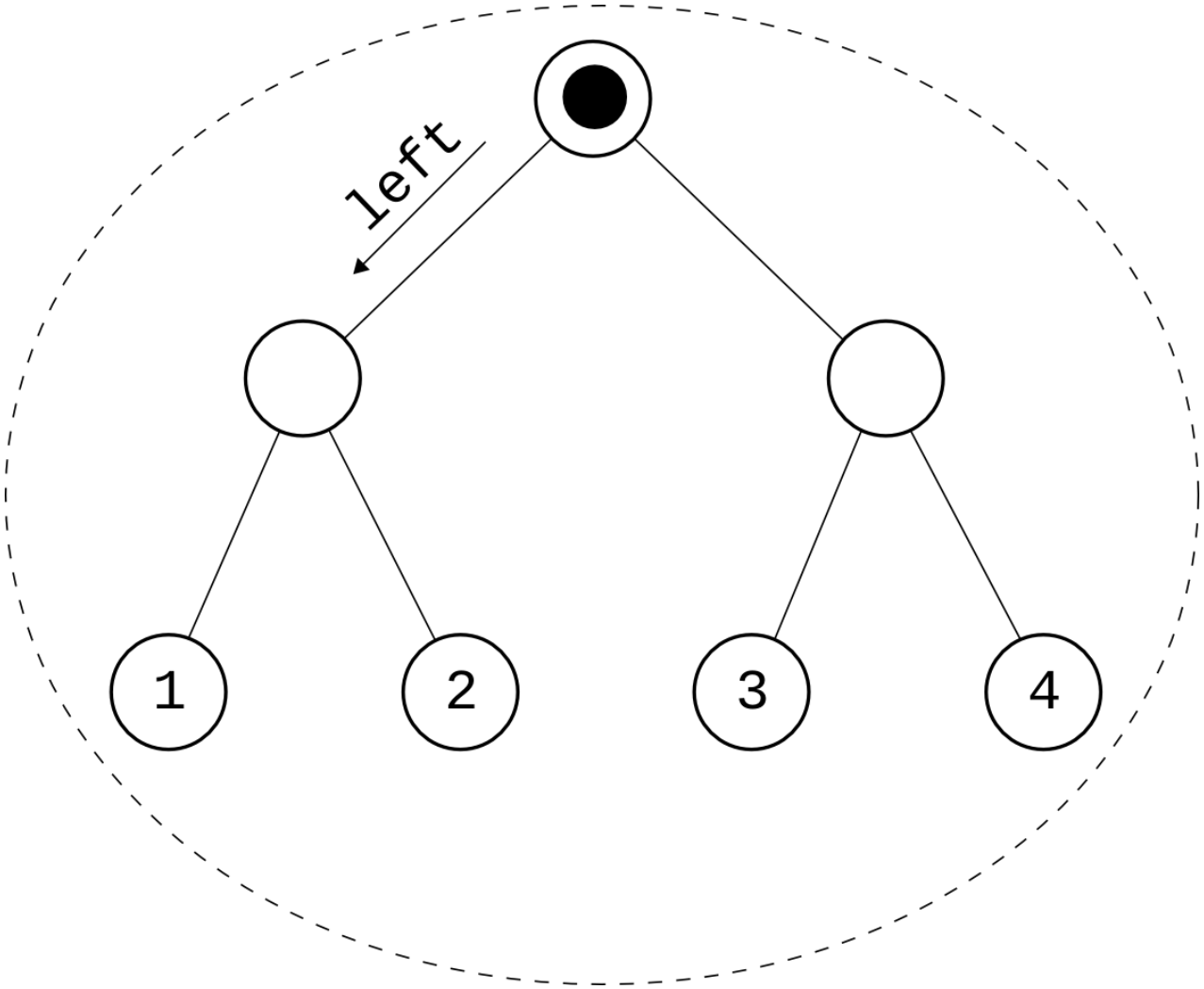
$$
\begin{aligned}
&\mathit{modify} :: (\alpha \to \mathit{Maybe}\,\alpha) \to \mathit{ZippList}\,\alpha \to \mathit{ZippList}\,\alpha \\
&\mathit{modify}\ f\ (\mathit{ZPL}\,(\mathit{xs},\ y : \mathit{ys})) = \mathit{ZPL}\ \$ \\
&\quad \textbf{case}\ (f\ y)\ \textbf{of} \\
&\qquad \mathit{Nothing} \to (\mathit{xs},\ \mathit{ys}) \\
&\qquad \mathit{Just}\ e \quad \to (\mathit{xs},\ e : \mathit{ys}) \\
&\mathit{modify}\ \_\ \mathit{zl} = \mathit{zl}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{update}\ f = \mathit{modify}\ (\mathit{Just} \circ f) \\
&\mathit{delete} \quad = \mathit{modify}\ (\mathit{const}\ \mathit{Nothing})
\end{aligned}
$$

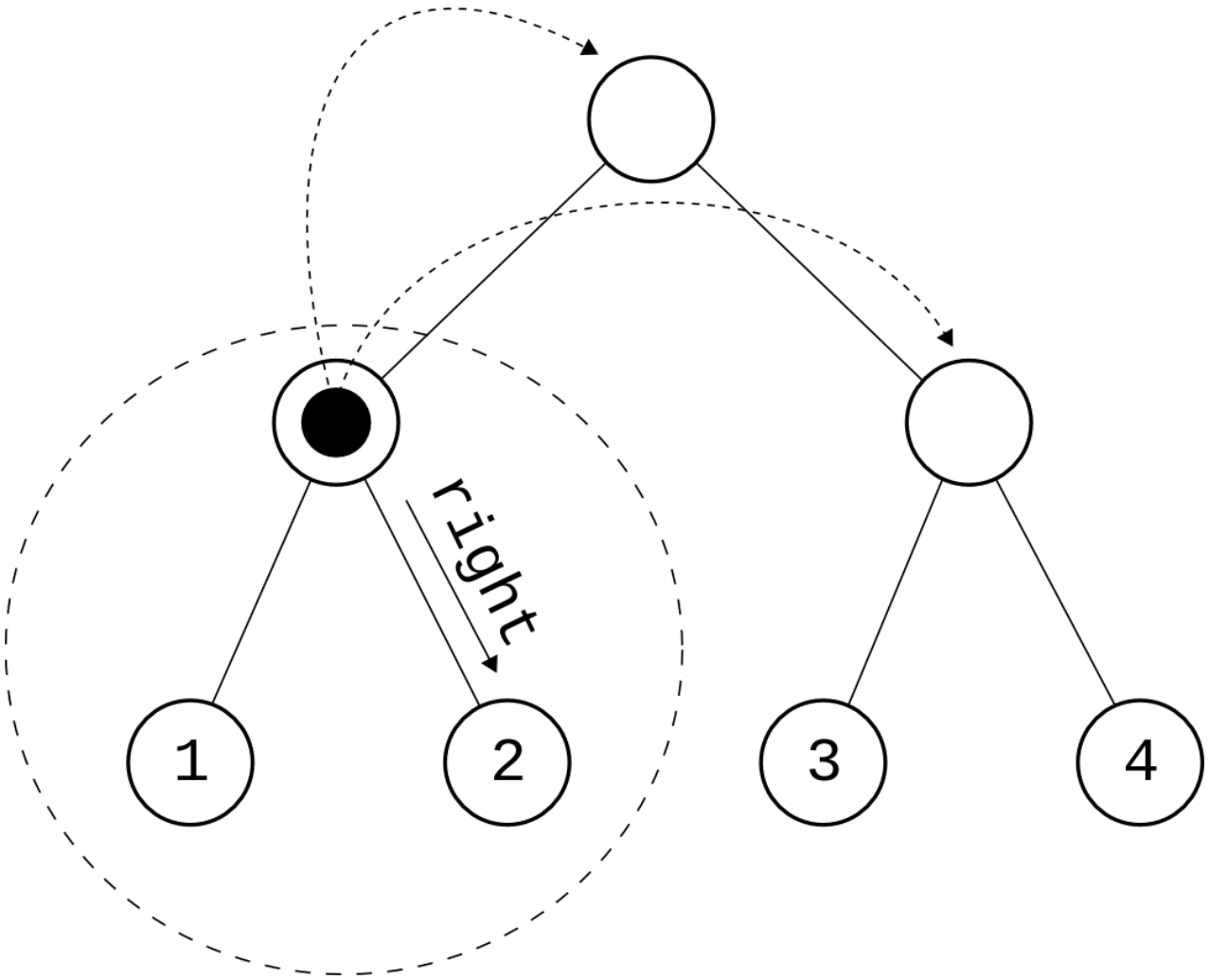
Kétirányú fák: "Tree with Zipper"



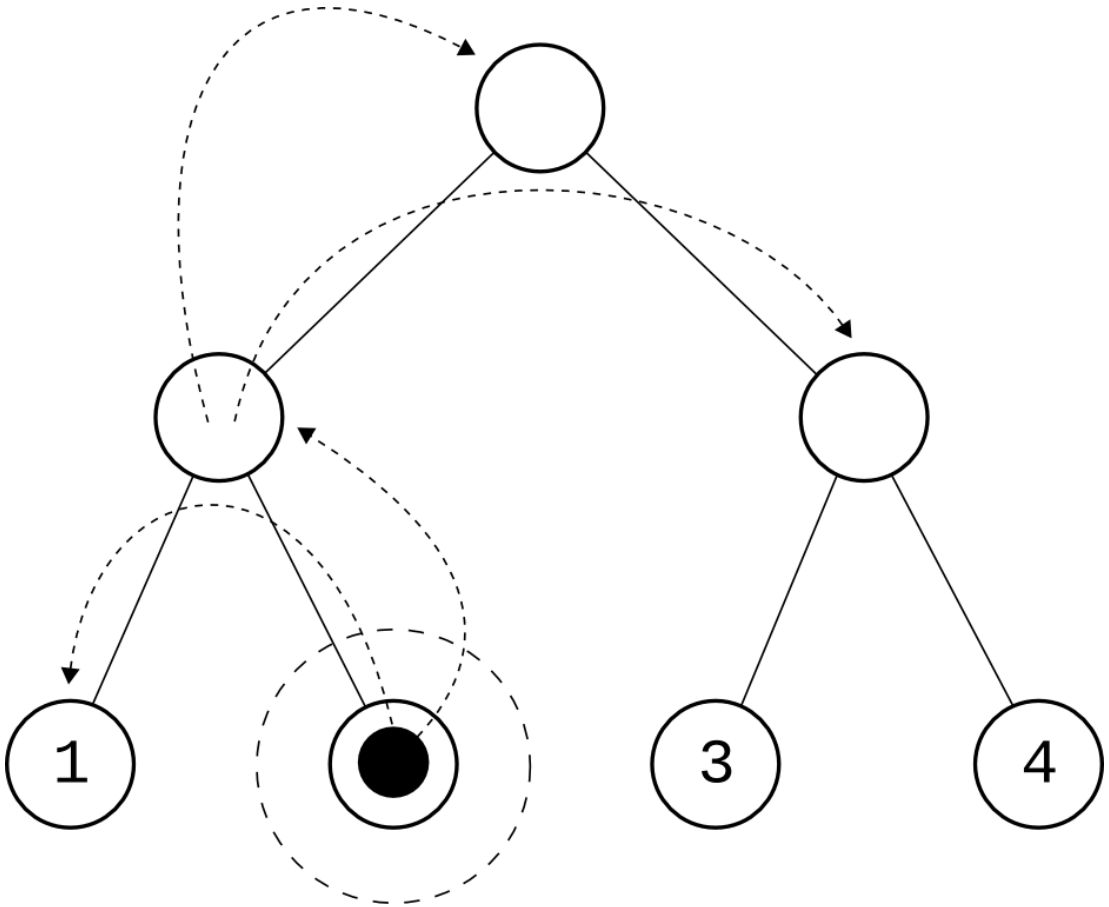
Kétirányú fák: "Tree with Zipper"



Kétirányú fák: "Tree with Zipper"



Kétirányú fák: "Tree with Zipper"



Kétirányú fák: implementáció (1)

$$
\begin{aligned}
&\textbf{data}\ \mathit{Tree}\ \alpha \\
&\quad = \mathit{Branch}\ (\mathit{Tree}\ \alpha)\ (\mathit{Tree}\ \alpha) \\
&\quad \mid\ \ \mathit{Leaf}\ \alpha \\
&\textbf{data}\ \mathit{TreeContext}\ \alpha \\
&\quad = \mathit{Top} \\
&\quad \mid\ \ \mathit{L}\ (\mathit{TreeContext}\ \alpha)\ (\mathit{Tree}\ \alpha) \\
&\quad \mid\ \ \mathit{R}\ (\mathit{Tree}\ \alpha)\ (\mathit{TreeContext}\ \alpha) \\
&\textbf{type}\ \mathit{Location}\ \alpha\ \gamma\ =\ (\alpha,\ \gamma) \\
&\textbf{type}\ \mathit{TreeLocation}\ \alpha\ =\ \mathit{Location}\ (\mathit{Tree}\ \alpha)\ (\mathit{TreeContext}\ \alpha)
\end{aligned}
$$

Kétirányú fák: implementáció (2)

$$
\begin{aligned}
&\mathit{treeLeft} :: \mathit{TreeLocation}\,\alpha \rightarrow \mathit{TreeLocation}\,\alpha \\
&\mathit{treeLeft}\,(\mathit{Branch}\,l\,r,\ c) = (l,\ L\,c\,r)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{treeRight} :: \mathit{TreeLocation}\,\alpha \rightarrow \mathit{TreeLocation}\,\alpha \\
&\mathit{treeRight}\,(\mathit{Branch}\,l\,r,\ c) = (r,\ R\,l\,c)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{treeTop} :: \mathit{Tree}\,\alpha \rightarrow \mathit{TreeLocation}\,\alpha \\
&\mathit{treeTop}\,t = (t,\ \mathit{Top})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{treeUp} :: \mathit{TreeLocation}\,\alpha \rightarrow \mathit{TreeLocation}\,\alpha \\
&\mathit{treeUp}\,(t,\ L\,c\,r) = (\mathit{Branch}\,t\,r,\ c) \\
&\mathit{treeUp}\,(t,\ R\,l\,c) = (\mathit{Branch}\,l\,t,\ c)
\end{aligned}
$$

Kétirányú fák: implementáció (3)

$$
\begin{aligned}
&\mathit{treeUpmost} :: \mathit{TreeLocation}\,\alpha \to \mathit{TreeLocation}\,\alpha \\
&\mathit{treeUpmost}\ l@(t,\ \mathit{Top}) = l \\
&\mathit{treeUpmost}\ l \qquad\qquad\quad = \mathit{treeUpmost}\,(\mathit{treeUp}\ l)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{treeChange} :: \mathit{TreeLocation}\,\alpha \to (\mathit{Tree}\,\alpha \to \mathit{Tree}\,\alpha) \\
&\qquad\qquad\qquad\quad \to \mathit{TreeLocation}\,\alpha \\
&\mathit{treeChange}\,(t,\ c)\ f = (f\ t,\ c)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{treeView} :: \mathit{TreeLocation}\,\alpha \to \mathit{Tree}\,\alpha \\
&\mathit{treeView}\,(t,\ \_) = t
\end{aligned}
$$

Például:

$$
\begin{aligned}
&\mathit{treeChange}\,((\mathit{treeRight} \circ \mathit{treeLeft} \circ \mathit{treeTop})\ t) \\
&\quad (\mathit{const}\,(\mathit{Leaf}\ 0))
\end{aligned}
$$

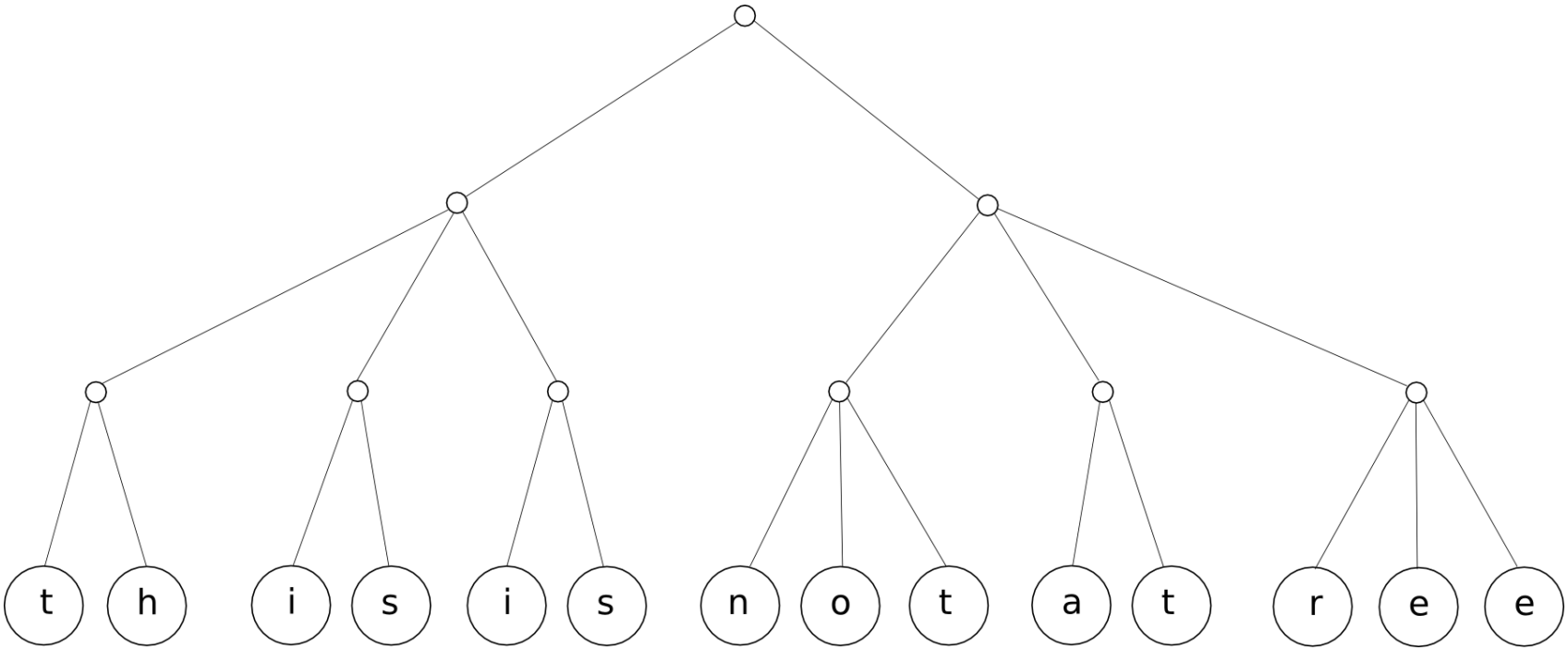
Ráadás: *Zipper* monád

```
newtype Zipper λ α = Zipper { unZipper :: State λ α }
  deriving (Functor, Applicative, Monad, MonadState λ)

traverse :: Location α γ → Zipper (Location α γ) α → α
traverse start tt = evalState (unZipper tt) start

move :: (Location α γ → Location α γ) → Zipper (Location α γ) α
move f = do
  modify f
  gets fst
change :: (α → α) → Zipper (Location α γ) α
change f = do
  modify (λ (t, c) . (f t, c))
  gets fst
```

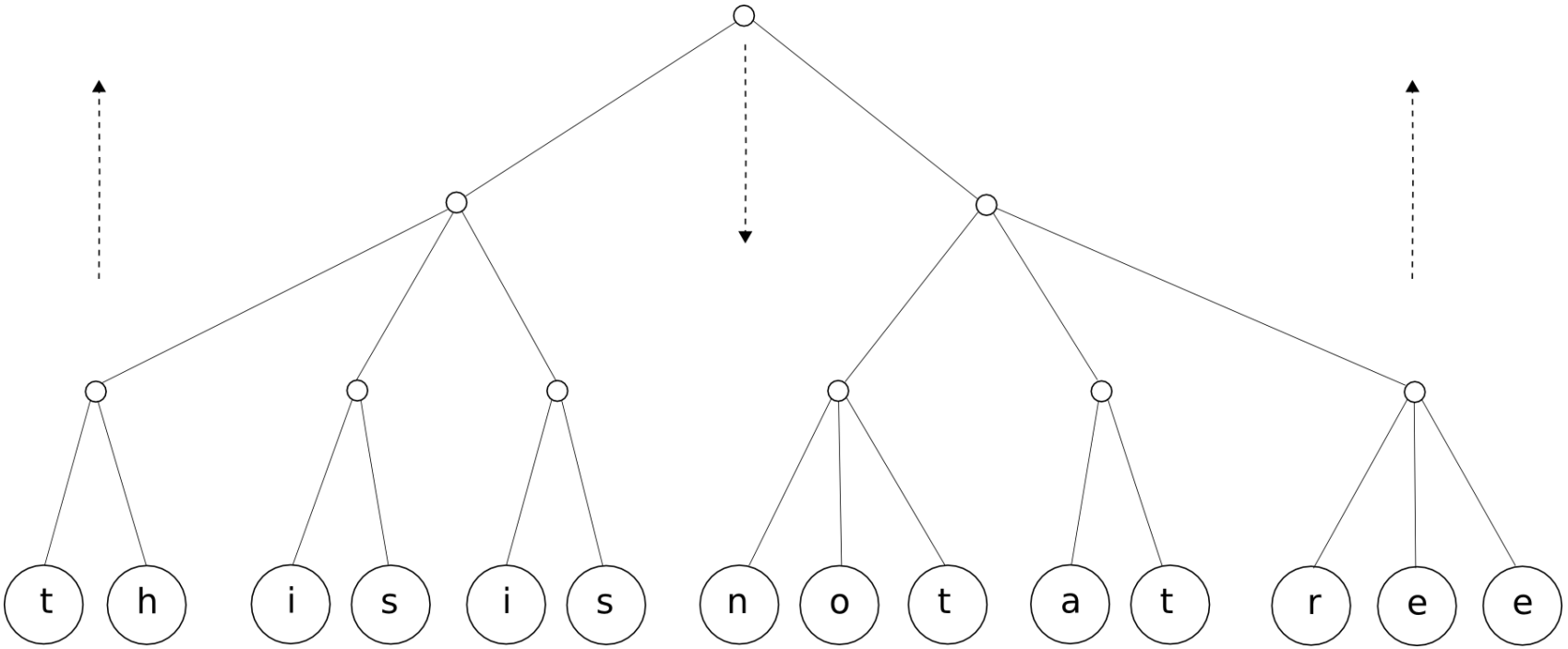
Zipper monád: példa

```
swapTree :: Zipper (TreeLocation α) (Tree α)
swapTree = move swap
  where
    swap (t, R l c) = (l, L c t)
    swap (t, L c r) = (r, R t c)

treeMap :: (α → Tree α) → (Tree α → Tree α → Tree α) → (Tree α → Tree α)
treeMap leaf branch = λ t . (treeTop t) `traverse` treeMapM
  where
    treeMapM = do
      t ← gets fst
      case t of
        Branch _ _ → do
          move treeLeft
          l_mod ← treeMapM
          swapTree
          r_mod ← treeMapM
          move treeUp
          (change ∘ const) (branch l_mod r_mod)
        Leaf x →
          return (leaf x)
```

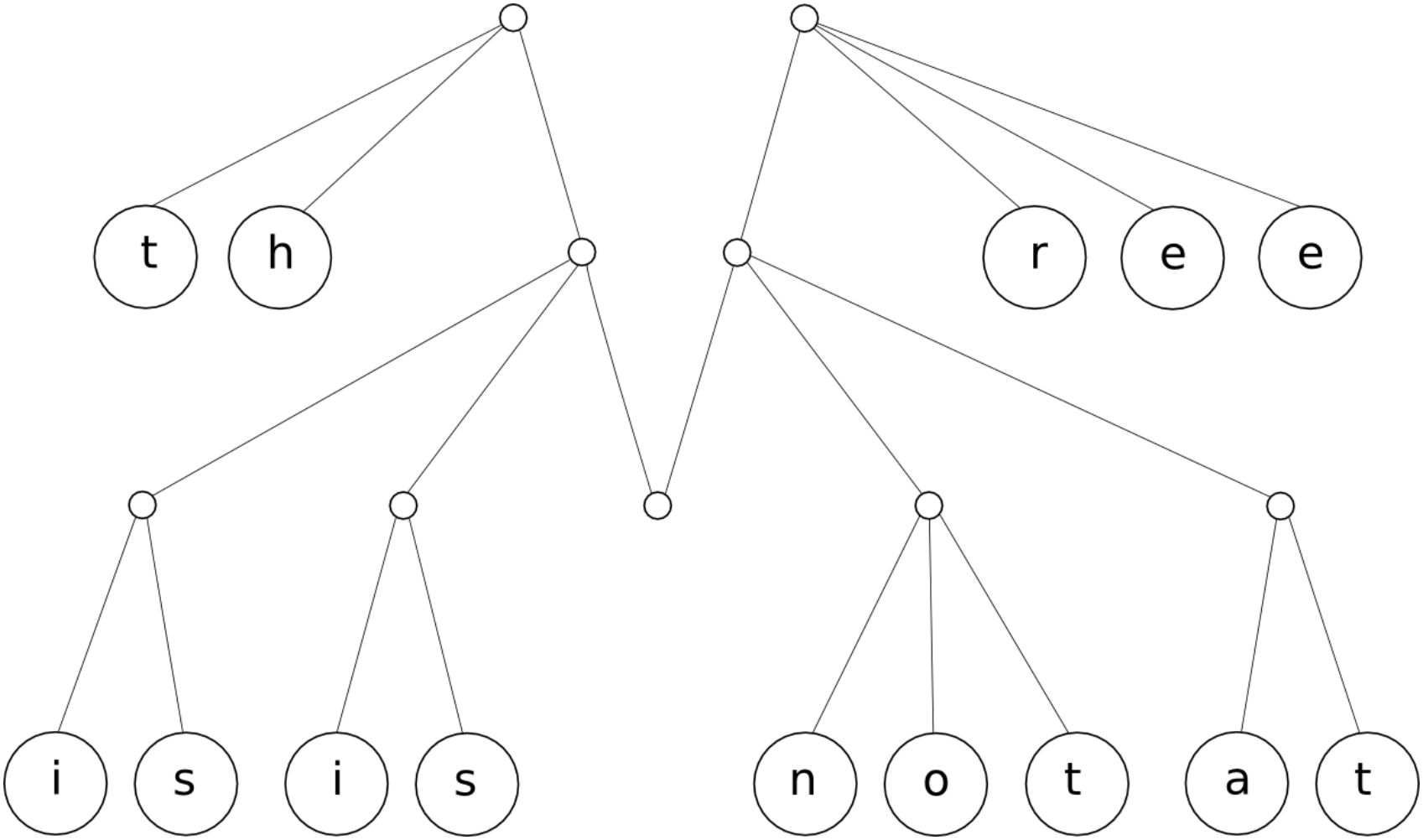
FingerTree (intuíció)



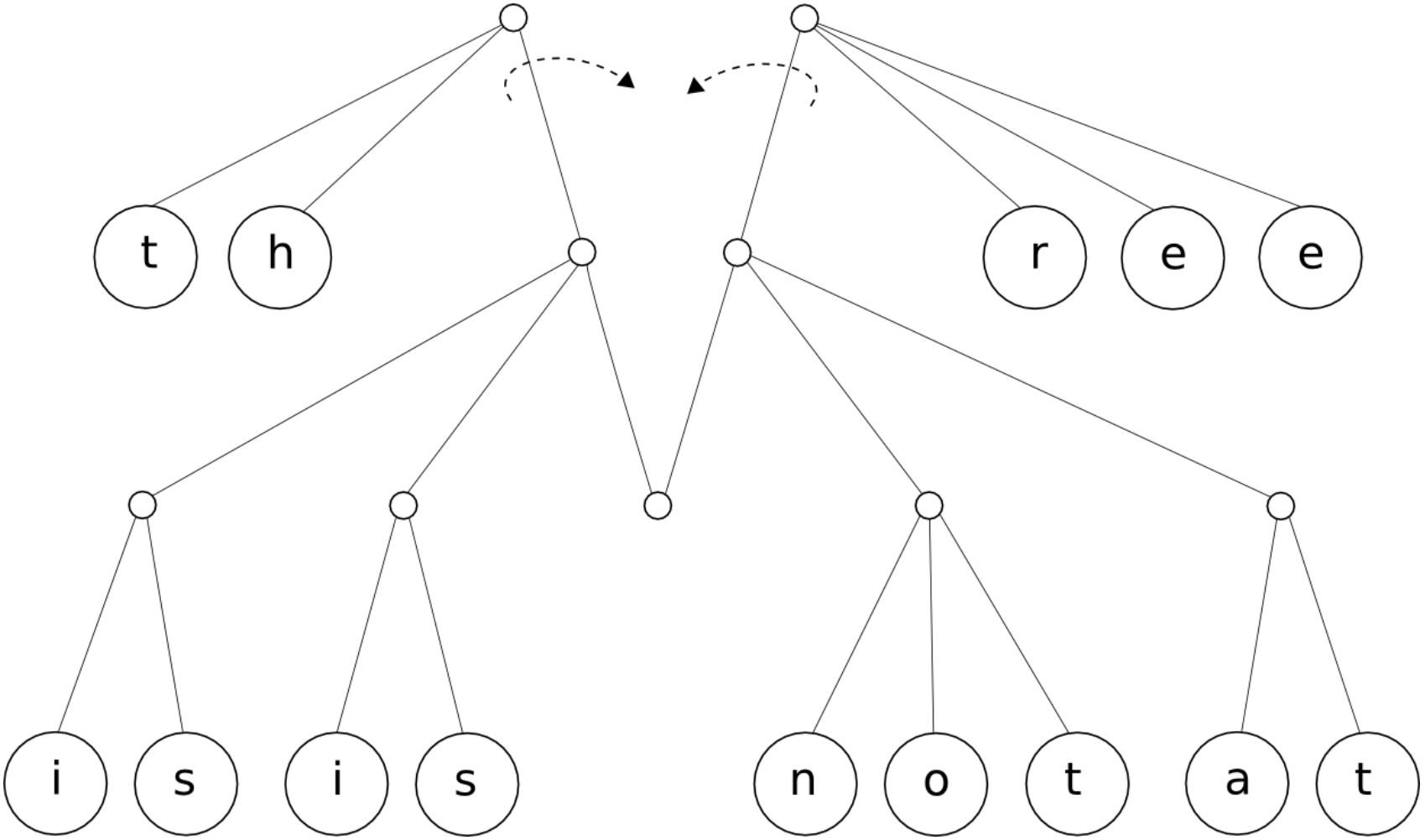
FingerTree (intuíció)



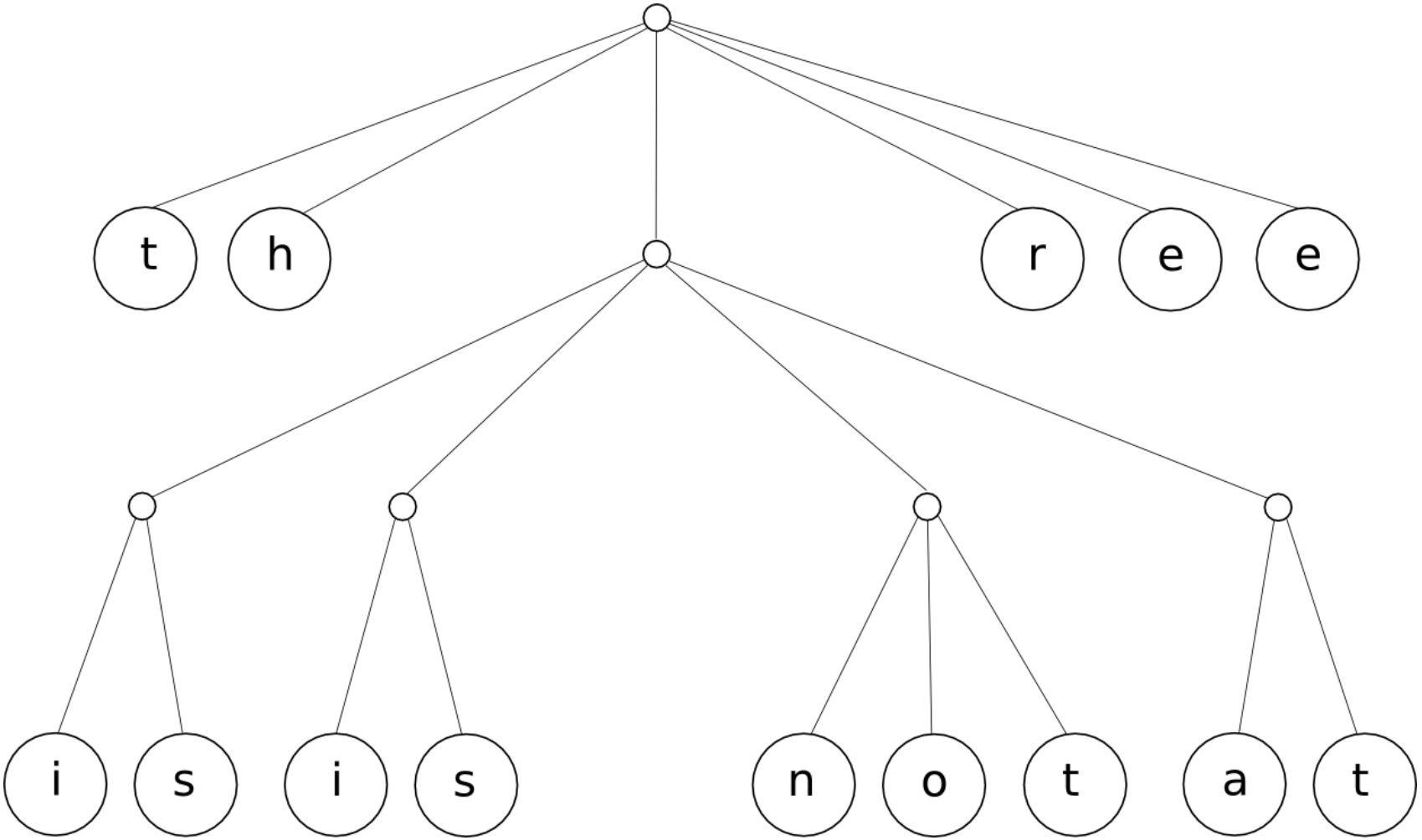
FingerTree (intuíció)



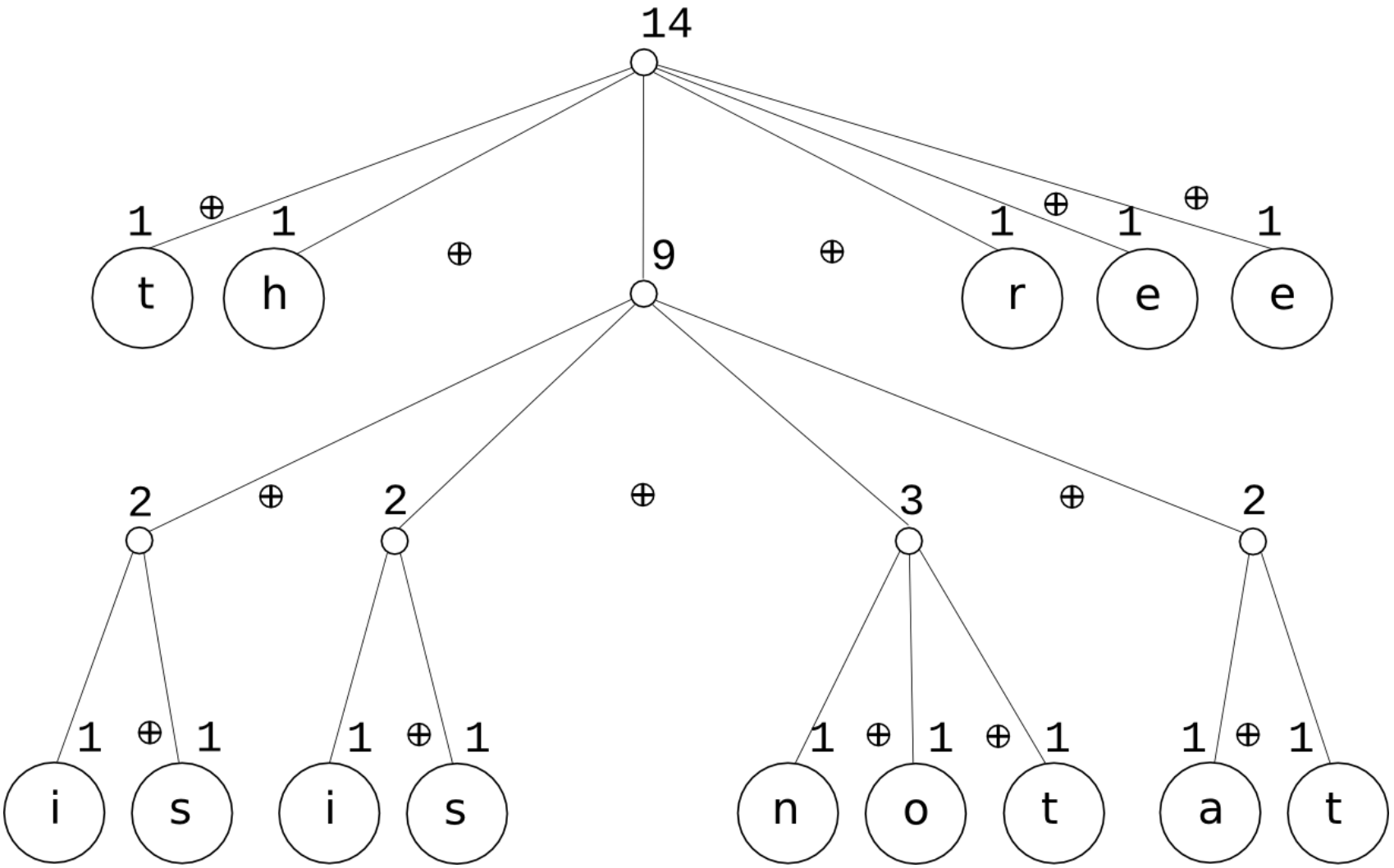
FingerTree (intuíció)



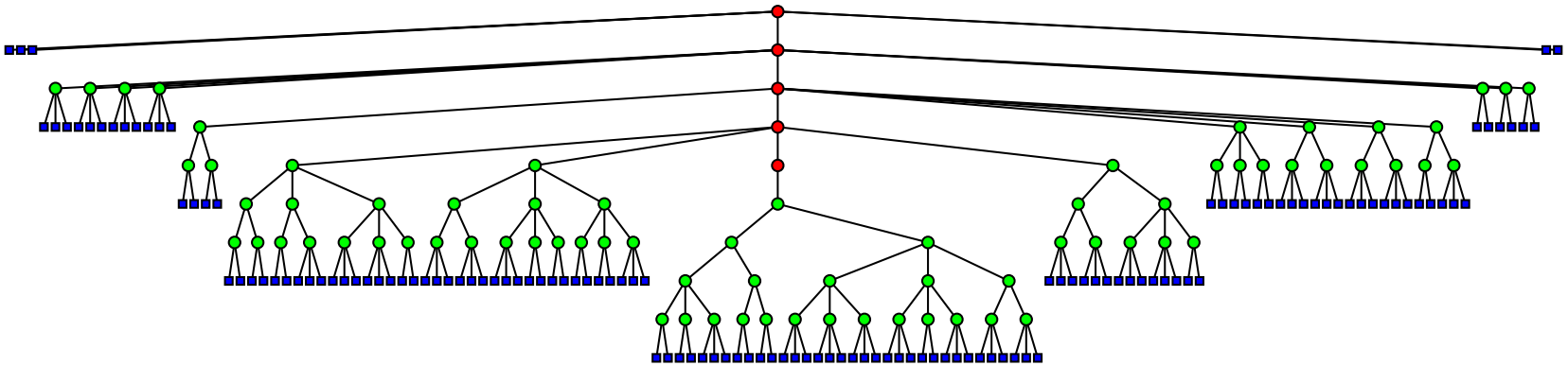
FingerTree (intuíció)



FingerTree (intuíció)



FingerTree (intuíció)



Implementáció: társított (indexelt) típusszinonimák

```
{-# LANGUAGE TypeFamilies, KindSignatures #-}
class Collects α where
  type Elem α :: ⋆

  empty  :: α
  insert :: Elem α → α → α
  ...

instance Eq (Elem [ε]) ⇒ Collects [ε] where
  type Elem [ε] = ε

  empty      = []
  insert e xs = (e : xs)
  ...
```

Implementáció: nézetminták

```
{-# LANGUAGE ViewPatterns #-}
type Typ = ...

data TypView = Unit | Arrow Typ Typ

view :: Typ → TypView
view = ...

size :: Typ → Integer
size t = case (view t) of
  Unit          → 1
  Arrow t1 t2   → size t1 + size t2
```

Nézetminták segítségével pedig:

```
size (view → Unit)          = 1
size (view → Arrow t1 t2)   = size t1 + size t2
```

Implementáció: mohón kiértékelt adatkonstruktorok

$$\textbf{data}\ T\ =\ T\ !Int\ !Int$$

- A konstruktor ! segítségével megjelölt paramétereit (strictness annotation) normálformára kell hozni, mielőtt azt alkalmazzuk.
- Körültekintéssel kell alkalmazni, mivel ez automatikusan nem vezet a teljesítmény növekedéséhez.
- Sőt, ronthatja a teljesítményt: ha az adott mezőt már egyszer kiértékeltük, akkor lényegében még egyszer kiértékeltetjük (feleslegesen).
- A helyzet tisztázásában a fordító nem mindig tud a segítségünkre lenni.