

# MGS 2012: FUN Lecture 2

## *Purely Functional Data Structures*

Henrik Nilsson

University of Nottingham, UK

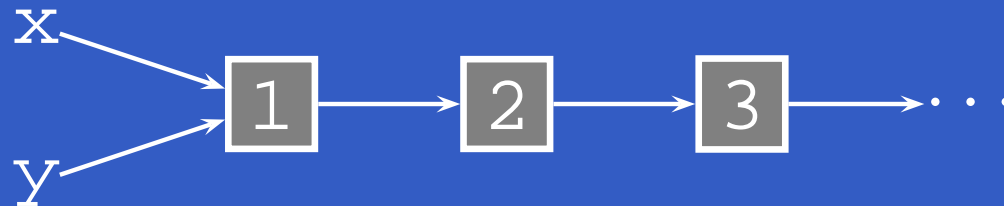
# Purely Functional Data structures (1)

Why is there a need to consider purely functional data structures?

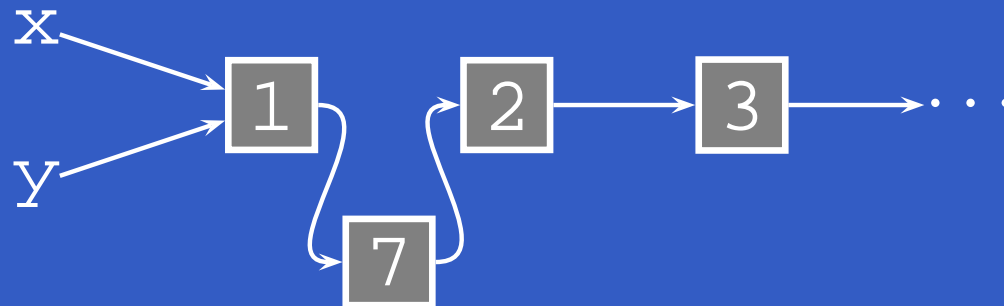
- The standard implementations of many data structures assume imperative update. To what extent truly necessary?
- Purely functional data structures are ***persistent***, while imperative ones are ***ephemeral***:
  - Persistence is a useful property in its own right.
  - Can't expect added benefits for free.

# Purely Functional Data structures (2)

Linked list:

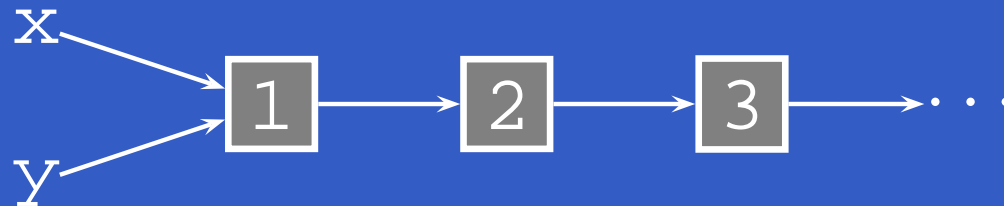


After insert, if ephemeral:

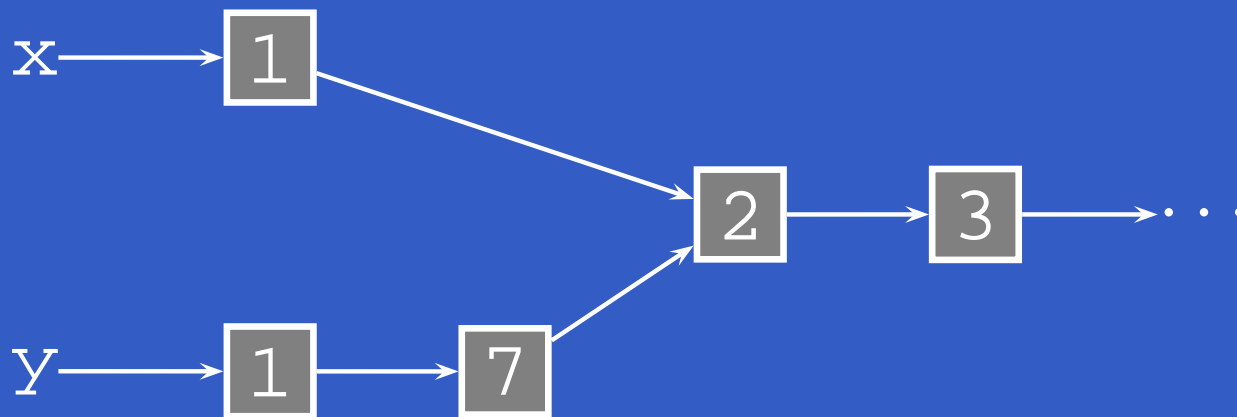


# Purely Functional Data structures (3)

Linked list:



After insert, if persistent:



# Purely Functional Data structures (4)

This lecture draws from:

Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

We will look at some examples of how **numerical representations** can be used to derive purely functional data structures.

# Numerical Representations (1)

Strong analogy between lists and the usual representation of natural numbers:

```
data List a =  
  Nil  
  | Cons a (List a)
```

```
data Nat =  
  Zero  
  | Succ Nat
```

```
tail (Cons _ xs) = xs
```

```
pred (Succ n) = n
```

```
append Nil ys = ys
```

```
plus Zero n = n
```

```
append (Cons x xs) ys =
```

```
plus (Succ m) n =
```

```
  Cons x (append xs ys)
```

```
  Succ (plus m n)
```

# Numerical Representations (2)

This analogy can be taken further for designing **container** structures because:

- inserting an element resembles incrementing a number
- combining two containers resembles adding two numbers

etc.

# Numerical Representations (2)

This analogy can be taken further for designing **container** structures because:

- inserting an element resembles incrementing a number
- combining two containers resembles adding two numbers

etc.

Thus, representations of natural numbers with certain properties **induce** container types with similar properties. Called **Numerical Representations**.



# Random Access Lists

We will consider *Random Access Lists* in the following. Signature:

```
data RList a
```

```
empty    :: RList a
```

```
isEmpty  :: RList a -> Bool
```

```
cons     :: a -> RList a -> RList a
```

```
head     :: RList a -> a
```

```
tail     :: RList a -> RList a
```

```
lookup   :: Int -> RList a -> a
```

```
update   :: Int -> a -> RList a  
          -> RList a
```

# Positional Number Systems (1)

- A number is written as a **sequence** of **digits**  $b_0b_1 \dots b_{m-1}$ , where  $b_i \in D_i$  for a fixed family of digit sets given by the positional system.
- $b_0$  is the **least significant** digit,  $b_{m-1}$  the **most significant** digit (note the ordering).
- Each digit  $b_i$  has a **weight**  $w_i$ . Thus:

$$\text{value}(b_0b_1 \dots b_{m-1}) = \sum_0^{m-1} b_i w_i$$

where the fixed sequence of weights  $w_i$  is given by the positional system.

# Positional Number Systems (2)

- A number is written in **base**  $B$  if  $w_i = B^i$  and  $D_i = \{0, \dots, B - 1\}$ .
- The sequence  $w_i$  is usually, but not necessarily, increasing.
- A number system is **redundant** if there is more than one way to represent some numbers (disallowing trailing zeroes).
- A representation of a positional number system can be **dense**, meaning including zeroes, or **sparse**, eliding zeroes.

# Exercise 1: Positional Number Systems

Suppose  $w_i = 2^i$  and  $D_i = \{0, 1, 2\}$ . Give three different ways to represent 17.

# Exercise 1: Solution

- 10001, since  $\text{value}(10001) = 1 \cdot 2^0 + 1 \cdot 2^4$
- 1002, since  $\text{value}(1002) = 1 \cdot 2^0 + 2 \cdot 2^3$
- 1021, since  $\text{value}(1021) = 1 \cdot 2^0 + 2 \cdot 2^2 + 1 \cdot 2^3$
- 1211, since  
 $\text{value}(1211) = 1 \cdot 2^0 + 2 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3$

# From Positional System to Container

Given a positional system, a numerical representation may be derived as follows:

- for a container of size  $n$ , consider a representation  $b_0b_1 \dots b_{m-1}$  of  $n$ ,
- represent the collection of  $n$  elements by a **sequence of trees** of size  $w_i$  such that there are  $b_i$  trees of that size.

For example, given the positional system of exercise 1, a container of size 17 might be represented by 1 tree of size 1, 2 trees of size 2, 1 tree of size 4, and 1 tree of size 8.

# What Kind of Trees?

The kind of tree should be chosen depending on needed sizes and properties. Two possibilities:

- **Complete Binary Leaf Trees**

```
data Tree a = Leaf a
             | Node (Tree a) (Tree a)
```

Sizes:  $2^n, n \geq 0$

- **Complete Binary Trees**

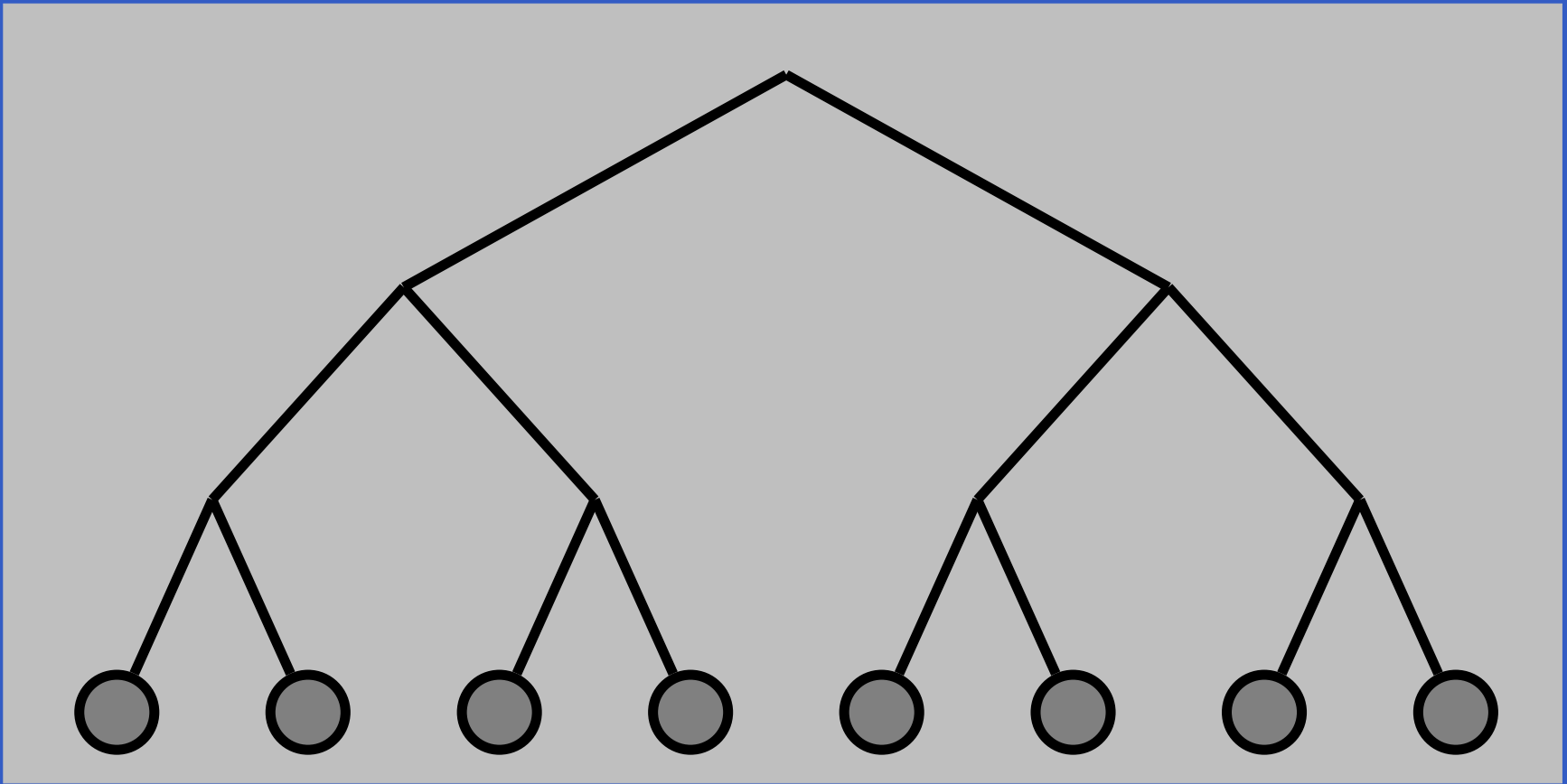
```
data Tree a = Leaf a
             | Node (Tree a) a (Tree a)
```

Sizes:  $2^{n+1} - 1, n \geq 0$

(Balance has to be ensured separately.)

# Example: Complete Binary Leaf Tree

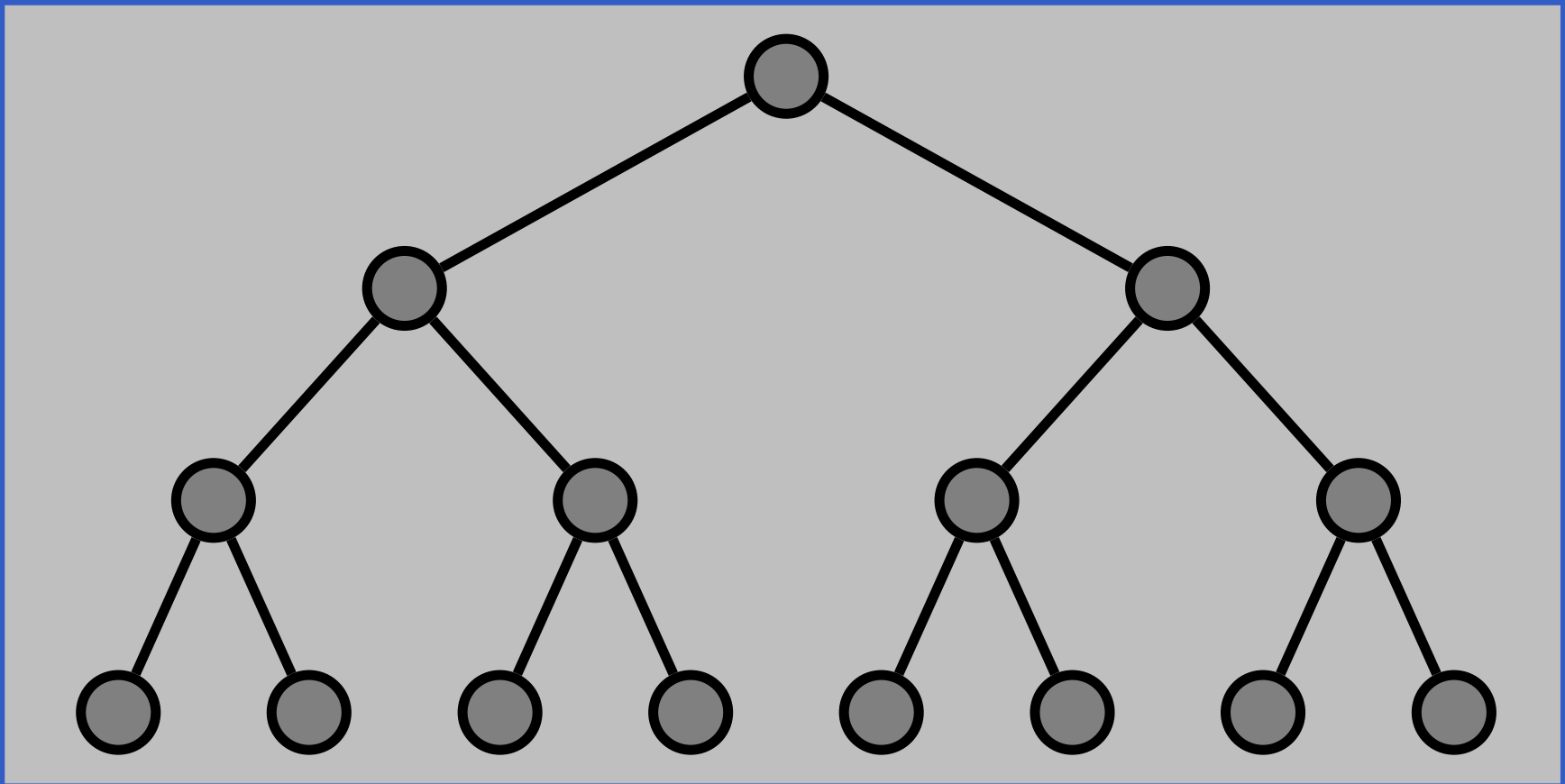
Size  $2^3 = 8$ :





# Example: Complete Binary Tree

Size  $2^4 - 1 = 15$ :



# Binary Random Access Lists (1)

**Binary Random Access Lists** are induced by

- the usual binary representation, i.e.  $w_i = 2^i$ ,  
 $D_i = \{0, 1\}$
- complete binary leaf trees

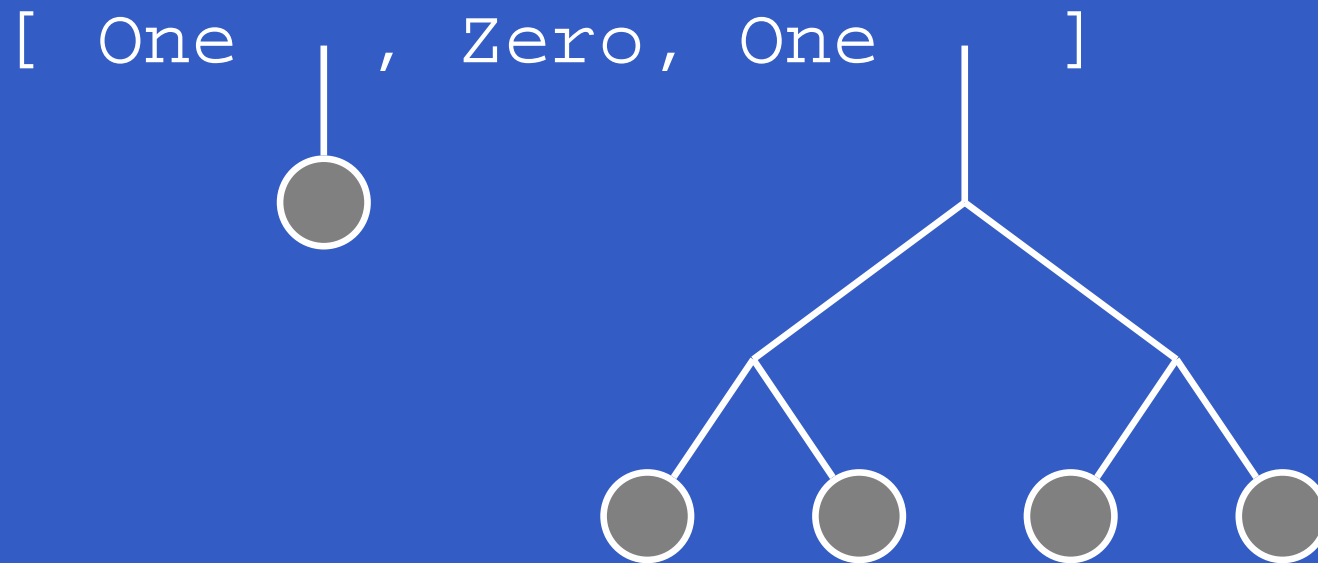
Thus:

```
data Tree a = Leaf a
              | Node Int (Tree a) (Tree a)
data Digit a = Zero | One (Tree a)
type RList a = [Digit a]
```

The `Int` field keeps track of tree size for speed.

# Binary Random Access Lists (2)

Example: Binary Random Access List of size 5:



# Binary Random Access Lists (3)

The increment function on dense binary numbers:

```
inc [] = [One]
```

```
inc (Zero : ds) = One : ds
```

```
inc (One : ds) = Zero : inc ds -- Carry
```

# Binary Random Access Lists (4)

Inserting an element first in a binary random access list is analogous to `inc`:

```
cons :: a -> RList a -> RList a
cons x ts = consTree (Leaf x) ts
```

```
consTree :: Tree a -> RList a -> RList a
consTree t [] = [One t]
consTree t (Zero : ts) = (One t : ts)
consTree t (One t' : ts) =
  Zero : consTree (link t t') ts
```

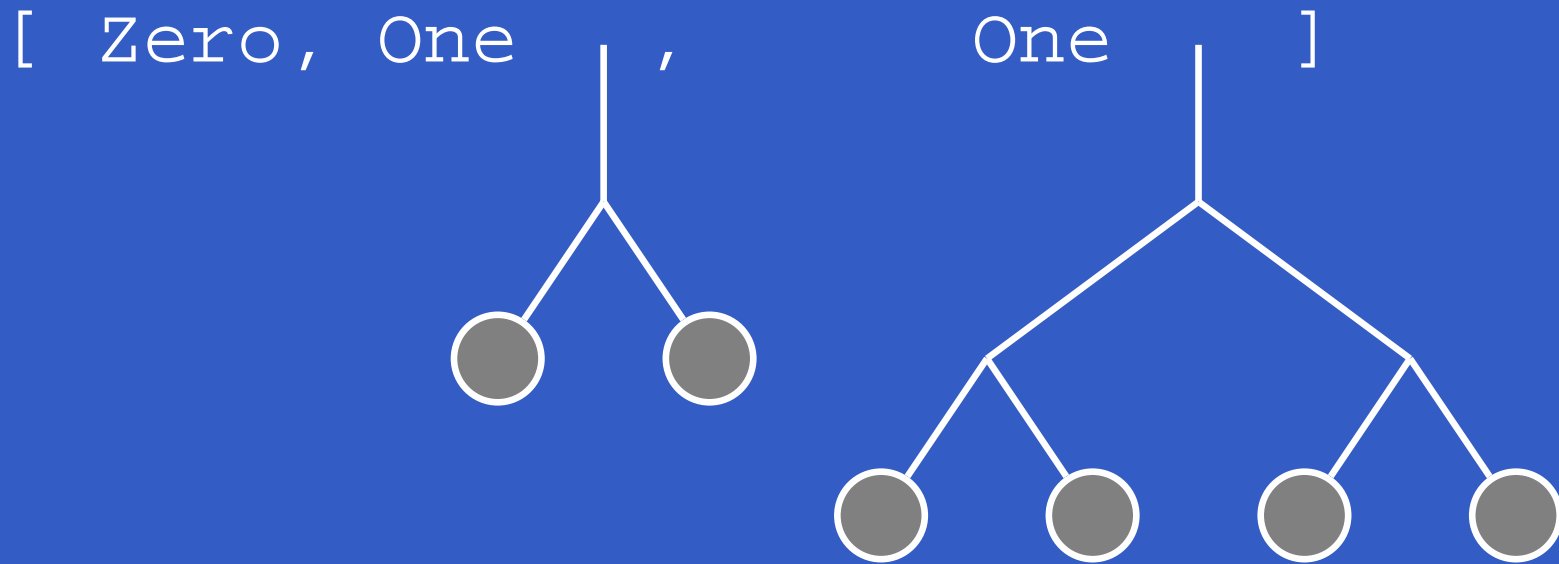
# Binary Random Access Lists (5)

The utility function `link` joins two equally sized trees:

```
-- t1 and t2 are assumed to be the same size
link t1 t2 = Node (2 * size t1) t1 t2
```

# Binary Random Access Lists (6)

Example: Result of consing element onto list of size 5:



## Exercise 2: unconstTree

The decrement function on dense binary numbers:

```
dec [One] = []
dec (One  : ds) = Zero  : ds
dec (Zero : ds) = One  : dec ds  -- Borrow
```

Define `unconstTree` following the above pattern:

```
unconstTree :: RList a -> (Tree a, RList a)
```

And then `head` and `tail`:

```
head :: RList a -> a
tail :: RList a -> RList a
```



# Exercise 2: Solution (1)

```
unconstTree :: RList a -> (Tree a, RList a)
unconstTree [One t]           = (t, [])
unconstTree (One t : ts)     = (t, Zero : ts)
unconstTree (Zero  : ts)     = (t1, One t2 : ts')
    where
        (Node _ t1 t2, ts') = unconstTree ts
```

Note: partial operation.

## Exercise 2: Solution (2)

```
head :: RList a -> a
```

```
head ts = x
```

```
  where
```

```
    (Leaf x, _) = unconstTree ts
```

```
tail :: RList a -> RList a
```

```
tail ts = ts'
```

```
  where
```

```
    (_, ts') = unconstTree ts
```

# Binary Random Access Lists (7)

Lookup is done in two stages: first find the right tree, then lookup in that tree:

```
lookup :: Int -> RList a -> a
lookup i (Zero : ts) = lookup i ts
lookup i (One t : ts)
  | i < s      = lookupTree i t
  | otherwise  = lookup (i - s) ts
  where
    s = size t
```

Note: partial operation.

# Binary Random Access Lists (8)

```
lookupTree :: Int -> Tree a -> a
lookupTree _ (Leaf x) = x
lookupTree i (Node w t1 t2)
  | i < w `div` 2 =
    lookupTree i t1
  | otherwise =
    lookupTree (i - w `div` 2) t2
```

The operation `update` has exactly the same structure.

# Binary Random Access Lists (9)

Time complexity:

- `cons`, `head`, `tail`, perform  $O(1)$  work per digit, thus  $O(\log n)$  worst case.
- `lookup` and `update` take  $O(\log n)$  to find the right tree, and then  $O(\log n)$  to find the right element in that tree, so  $O(\log n)$  worst case overall.

# Binary Random Access Lists (9)

Time complexity:

- `cons`, `head`, `tail`, perform  $O(1)$  work per digit, thus  $O(\log n)$  worst case.
- `lookup` and `update` take  $O(\log n)$  to find the right tree, and then  $O(\log n)$  to find the right element in that tree, so  $O(\log n)$  worst case overall.

Time complexity for `cons`, `head`, `tail`  
disappointing: can we do better?

# Skew Binary Numbers (1)

Skew Binary Numbers:

- $w_i = 2^{i+1} - 1$  (rather than  $2^i$ )
- $D_i = \{0, 1, 2\}$

Representation is redundant. But we obtain a **canonical form** if we insist that only the least significant non-zero digit may be 2.

Note: The weights correspond to the sizes of **complete** binary trees.

# Skew Binary Numbers (2)

Theorem: Every natural number  $n$  has a unique skew binary canonical form.

Proof *sketch*. By induction on  $n$ .

- Base case: the case for 0 is direct.



# Skew Binary Numbers (3)

- Inductive case. Assume  $n$  has a unique skew binary representation  $b_0b_1 \dots b_{m-1}$

# Skew Binary Numbers (3)

- Inductive case. Assume  $n$  has a unique skew binary representation  $b_0b_1 \dots b_{m-1}$ 
  - If the least significant non-zero digit is smaller than 2, then  $n + 1$  has a unique skew binary representation obtained by adding 1 to the least significant digit  $b_0$ .

# Skew Binary Numbers (3)

- Inductive case. Assume  $n$  has a unique skew binary representation  $b_0b_1 \dots b_{m-1}$ 
  - If the least significant non-zero digit is smaller than 2, then  $n + 1$  has a unique skew binary representation obtained by adding 1 to the least significant digit  $b_0$ .
  - If the least significant non-zero digit  $b_i$  is 2, then note that  $1 + 2(2^{i+1} - 1) = 2^{i+2} - 1$ . Thus  $n + 1$  has a unique skew binary representation obtained by setting  $b_i$  to 0 and adding 1 to  $b_{i+1}$ .

# Exercise 3a: Skew Binary Numbers

- Give the canonical skew binary representation for 31, 30, 29, and 28.

# Exercise 3a: Skew Binary Numbers

- Give the canonical skew binary representation for 31, 30, 29, and 28.
- Solution: 00001, 0002, 0021, 0211

# Exercise 3b: Skew Binary Numbers

Assume a **sparse** skew binary representation of the natural numbers

```
type Nat = [Int]
```

where the integers represent the **weight** of each **non-zero** digit. Assume further that the integers are stored in increasing order, except that the first two may be equal indicating that the smallest non-zero digit is 2. E.g.  $28 = [3, 3, 7, 15]$ .

Implement a function `inc` to increment a natural number. (E.g. `inc [3, 3, 7, 15] = [7, 7, 15]`)

# Exercise 3b: Solution

```
inc :: Nat -> Nat
inc (w1 : w2 : ws)
  | w1 == w2 = w1 * 2 + 1 : ws
inc ws      = 1 : ws
```

***Note! No carry propagation!***

E.g.:

```
inc [1,3,7,15] = [1,1,3,7,15]
inc [1,1,3,7,5] = [3,3,7,15]
inc [3,3,7,15] = [7,7,15]
```

# Skew Binary Random Access Lists (1)

```
data Tree a = Leaf a | Node (Tree a) a (Tree a)
type RList a = [(Int, Tree a)]
```

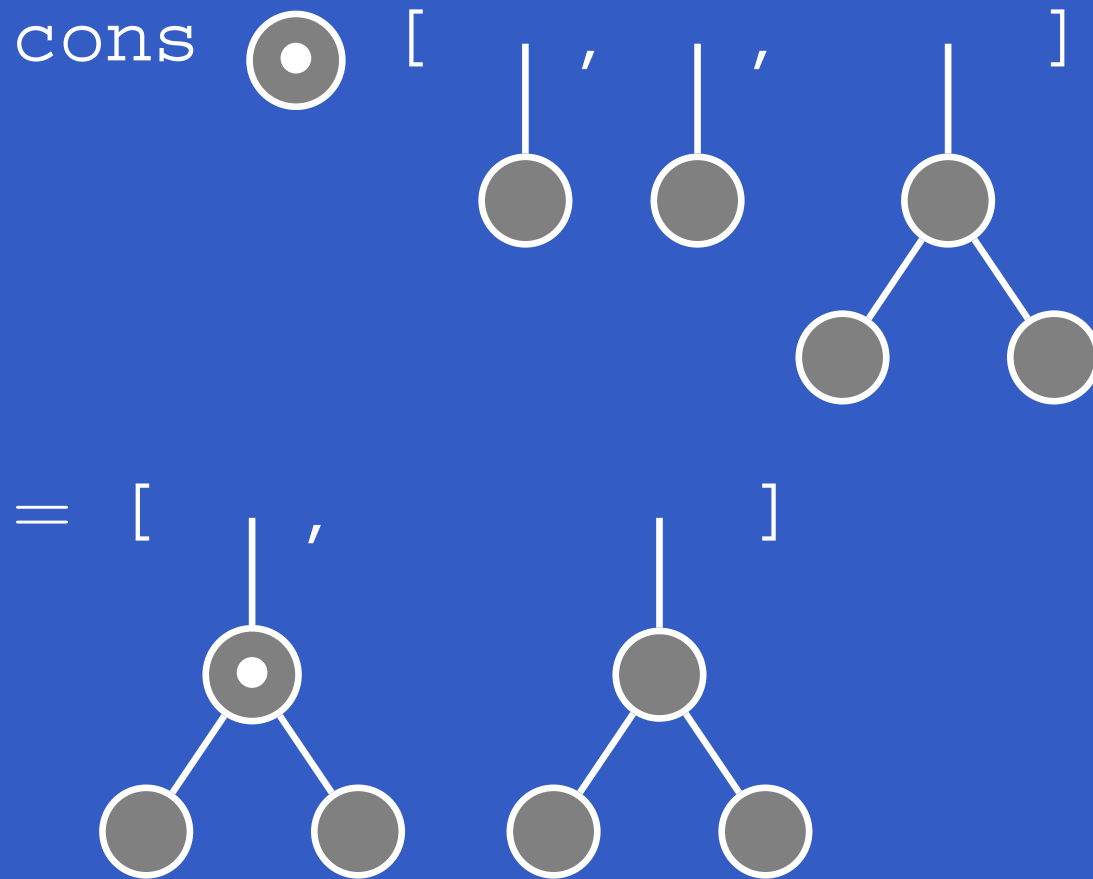
```
empty :: RList a
empty = []
```

```
cons :: a -> RList a -> RList a
cons x ((w1, t1) : (w2, t2) : wts) | w1 == w2 =
    (w1 * 2 + 1, Node t1 x t2) : wts
cons x wts = ((1, Leaf x) : wts)
```



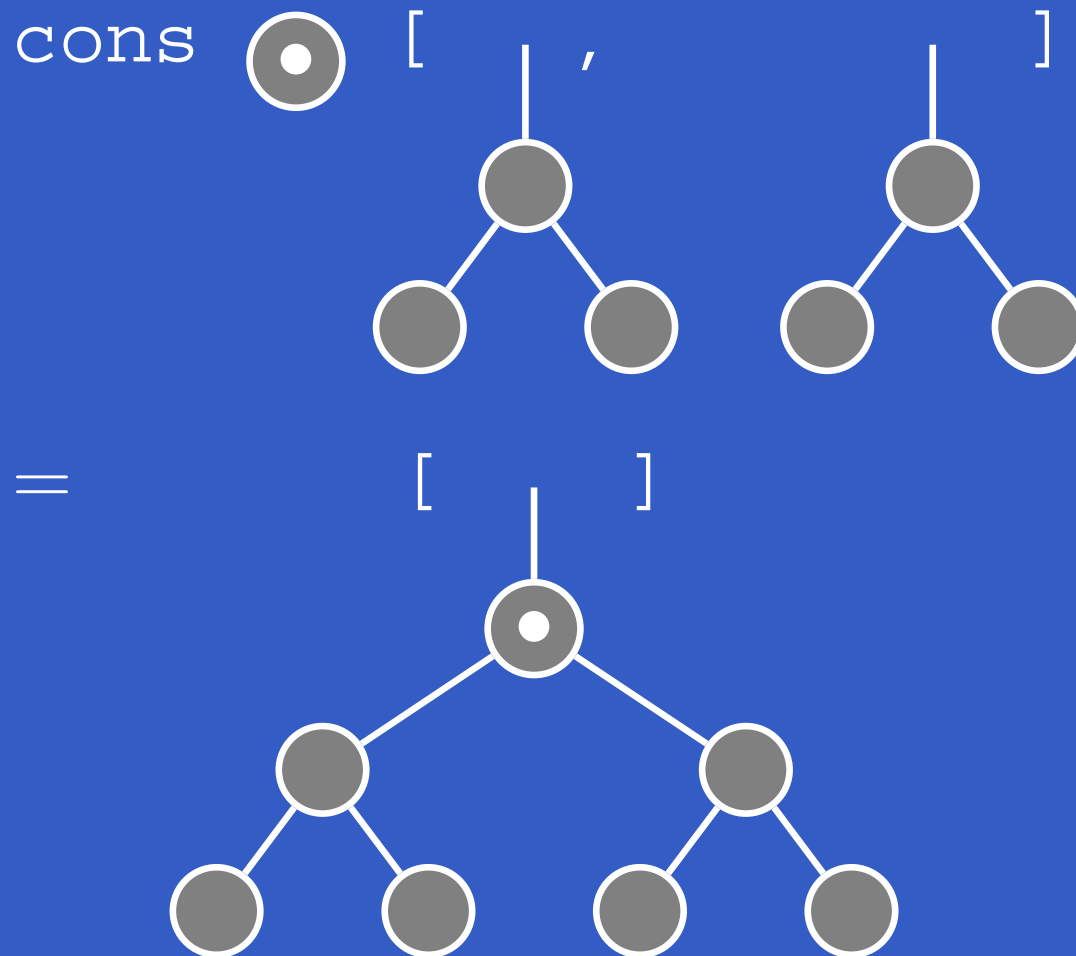
# Skew Binary Random Access Lists (2)

Example: Consing onto list of size 5:



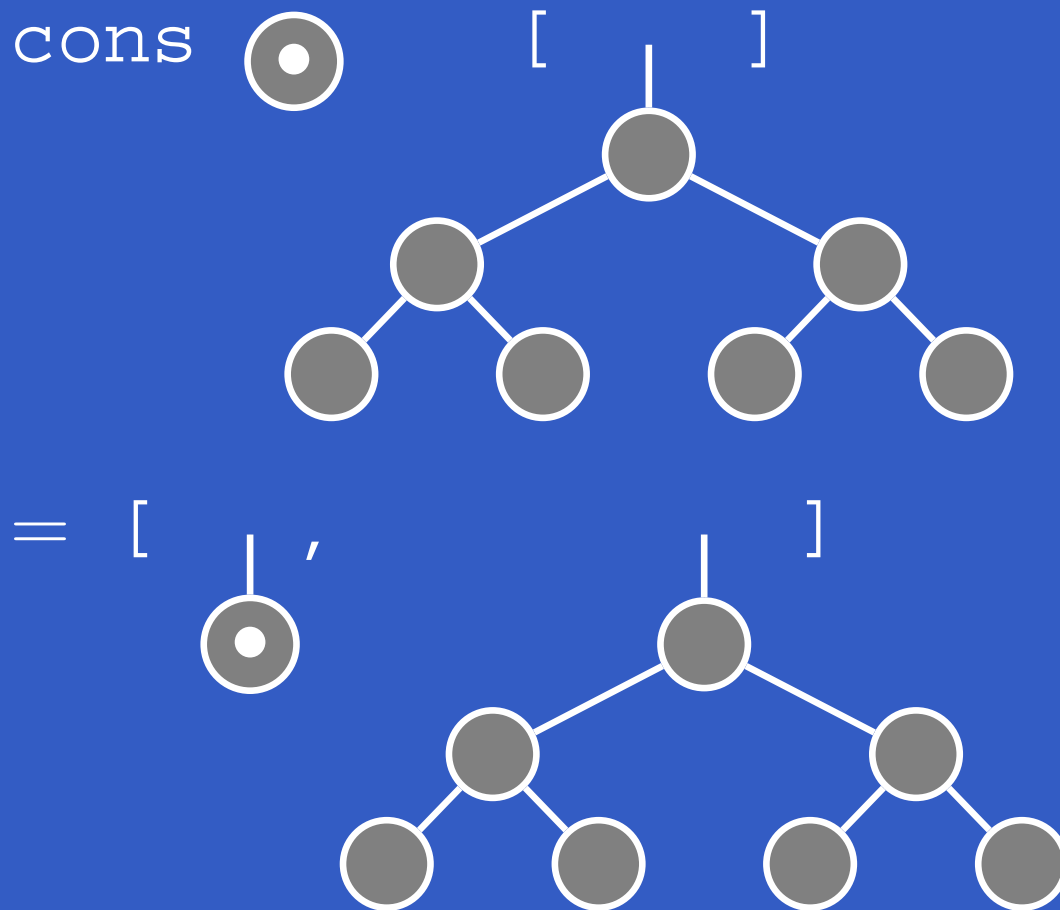
# Skew Binary Random Access Lists (3)

Example: Consing onto list of size 6:



# Skew Binary Random Access Lists (4)

Example: Consing onto list of size 7:



# Skew Binary Random Access Lists (5)

```
head :: RList a -> a
```

```
head ((_, Leaf x) : _) = x
```

```
head ((_, Node _ x _) : _) = x
```

```
tail :: RList a -> RList a
```

```
tail ((_, Leaf _) : wts) = wts
```

```
tail ((w, Node t1 _ t2) : wts) =  
    (w', t1) : (w', t2) : wts
```

where

```
w' = w `div` 2
```

Note: again, partial operations.

# Skew Binary Random Access Lists (6)

```
lookup :: Int -> RList a -> a
```

```
lookup i ((w, t) : wts)
```

```
  | i < w      = lookupTree i w t
```

```
  | otherwise  = lookup (i - w) wts
```

```
lookupTree :: Int -> Int -> Tree a -> a
```

```
lookupTree _ _ (Leaf x) = x
```

```
lookupTree i w (Node t1 x t2)
```

```
  | i == 0      = x
```

```
  | i < w'      = lookupTree (i - 1) w' t1
```

```
  | otherwise   = lookupTree (i - w' - 1) w' t2
```

where

```
  w' = w `div` 2
```

# Skew Binary Random Access Lists (7)

Time complexity:

- `cons`, `head`, `tail`:  $O(1)$ .
- `lookup` and `update` take  $O(\log n)$  to find the right tree, and then  $O(\log n)$  to find the right element in that tree, so  $O(\log n)$  worst case overall.

# Skew Binary Random Access Lists (7)

Time complexity:

- `cons`, `head`, `tail`:  $O(1)$ .
- `lookup` and `update` take  $O(\log n)$  to find the right tree, and then  $O(\log n)$  to find the right element in that tree, so  $O(\log n)$  worst case overall.

Okasaki:

“Although there are better implementations of lists, and better implementations of (persistent) arrays, none are better at both.”