# Lecture 6. Data structures

## Functional Programming 2018/19

Alejandro Serrano



Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

# Goals

Practice our Haskell skills

- Operations on binary trees
  - Common operations
  - Search trees
- Key-value maps
  - Via lists and via functions

# Binary search trees



Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

# Definition of `Tree`

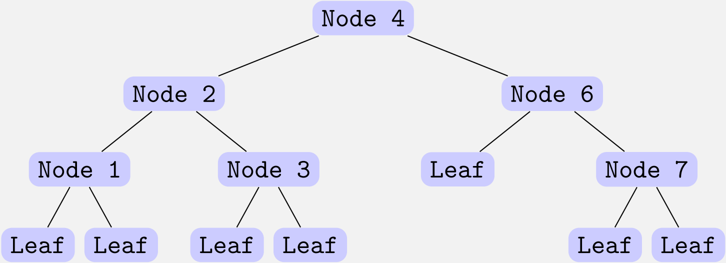
## Binary trees with data in the nodes

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

Figure 1:

# Example of tree

```
Node (Node (Node Leaf 1 Leaf)
           2
           (Node Leaf 3 Leaf))
     4
     (Node Leaf 6 (Node Leaf 7 Leaf))
```



# Other kinds of trees

- Binary trees with data in the leaves

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

- Binary trees with data in nodes and leaves
  - Potentially of different type

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

- Ternary trees with data in the nodes

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

# Rose trees

Trees with an unbound number of branches at each node

```
data RoseTree a = Leaf a
                | Node a [Tree a]
```

We do not really need `Leaf`, we can make the list empty

```
data RoseTree a = Node a [Tree a]
```

In the practicals, we use an infix constructor

```
data RoseTree a = a :> [Tree a]
```

# Cooking `size`

`size t` returns the number of (inner) nodes in `t`

1. Define the type

   ```
   size :: Tree a -> Int
   ```

2. Enumerate the cases

   ```
   size Leaf         = _
   size (Node l x r) = _
   ```

3. Define the cases

   - Each recursive position leads to a recursive call

   ```
   size Leaf         = 0
   size (Node l x r) = 1 + size l + size r
   ```

# Cooking `mirror`

`mirror t` returns the "mirror" image of `t`

```
> mirror (Node (Node Leaf 3 Leaf) 2 Leaf)
(Node Leaf 2 (Node Leaf 3 Leaf))
```

1. Define the type

```
mirror :: Tree a -> Tree a
```

2. Enumerate the cases

```
mirror Leaf         = _
mirror (Node l x r) = _
```

# Cooking `mirror`

`mirror t` returns the "mirror" image of `t`

```
> mirror (Node (Node Leaf 3 Leaf) 2 Leaf)
(Node Leaf 2 (Node Leaf 3 Leaf))
```

1. Define the type

   ```
   mirror :: Tree a -> Tree a
   ```

2. Enumerate the cases

   ```
   mirror Leaf         = _
   mirror (Node l x r) = _
   ```

3. Define the cases

   ```
   mirror Leaf         = Leaf
   mirror (Node l x r) = Node (mirror r) x (mirror l)
   ```

# Cooking `enumInfix`

`enumInfix t` returns the values of `t` in infix order

- From left-most to right-most
- The data in the node in between that of the subtrees

```
> enumInfix (Node (Node Leaf 2 Leaf) 3 Leaf)
[2,3]
```

1. Define the type

   ```
   enumInfix :: Tree a -> [a]
   ```

2. Enumerate the cases

   ```
   enumInfix Leaf         = _
   enumInfix (Node l x r) = _
   ```

# Cooking `enumInfix`

3. Define the simple (base) cases

```
enumInfix Leaf           = []
```

4. Define the other (recursive) cases

```
enumInfix (Node l x r) = enumInfix l
                         ++ [x]
                         ++ enumInfix r
```

   - Repeated calls to `(++)` are very expensive!
   - Solution: use an accumulator

# enumInfix with an accumulator

1. Introduce a local definition with an extra argument
2. Initialize the function in the main call

```
enumInfix t = enumInfix' t []
  where enumInfix' t acc = _
```

3. Follow Hutton's recipe, but
   - Do not pattern match on the accumulator
   - Return the accumulator in the base case
   - Update the accumulator in the recursive steps

```
enumInfix t = enumInfix' t []
  where enumInfix' Leaf acc = acc
        enumInfix' (Node l x r) acc
          = enumInfix' l (x : enumInfix' r acc)
```

# Linear search is expensive

```haskell
elem :: Eq a => a -> [a] -> Bool
elem _ []                    = False
elem e (x:xs) | e == x       = True
              | otherwise = elem e xs
```

- We check the elements one by one for equality
- If the element is not there, we make $n$ comparisons!
  - where $n$ is the length of the list
- On average, we make $\frac{n}{2}$ comparisons

*Technical note*: we say that linear search has $\mathcal{O}(n)$ complexity

# Linear search is expensive

Suppose that we guarantee that the input list is sorted

Can we make linear search better?

# Linear search in ordered lists

If we guarantee that the list is sorted, we can stop earlier

```
elem :: Ord a => a -> [a] -> Bool
elem _ []                    = False
elem e (x:xs) | e == x       = True
              | e <  x       = False  -- (!)
              | otherwise = elem e xs
```

Still, we look at all the elements before the one we search

- To do even better we need binary search


Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

# Search trees

Search trees are binary trees with a restriction over nodes

- All elements in the left subtree must be *smaller* than the data in the node
- Conversely, all elements in the right subtree must be *larger* than the data in the node

```haskell
-- Not a search tree, 3 > 2
Node (Node Leaf 3 Leaf) 2 (Node Leaf 4 Leaf)

-- A search tree with the same data
Node (Node Leaf 2 Leaf) 3 (Node Leaf 4 Leaf)
```

# Binary search

The ordering guides us on which subtree to consider

```
elem :: Ord a => a -> Tree a -> Bool
elem _ Leaf                     = False
elem e (Node l x r) | e == x = True
                    | e <  x = elem e l
                    | e >  x = elem e r
```

If the tree is "nicely built", we get $\mathcal{O}(\log n)$ complexity

# Building a search tree

We build the tree by repeated insertion

- `insert x t` adds the element `x` to the search tree `t`, respecting all the restrictions

```
toSearchTree :: Ord a => [a] -> Tree a
toSearchTree []     = Leaf
toSearchTree (x:xs) = insert x (toSearchTree xs)

-- Even better with a fold
toSearchTree :: Ord a => [a] -> Tree a
toSearchTree = foldr insert Leaf
```

# Cooking `insert`

1. Define the type

```
insert :: Ord a => a -> Tree a -> Tree a
```

# Cooking `insert`

1. Define the type

```
insert :: Ord a => a -> Tree a -> Tree a
```

2. Enumerate the cases

```
insert e Leaf           = _
insert e (Node l x r) = _
```

3. Define the simple (base) cases
   - If the tree is empty, we build one with the value

```
insert e Leaf = Node Leaf e Leaf
```

# Cooking `insert`

4. Define the other (recursive) cases
   - We need to compare the value with the node to decide where to continue
   - We prevent duplicates by an additional equality check

```
insert e (Node l x r)
  | e == x    = -- It's already there
                Node l              x r
  | e <  x    = Node (insert e l) x r
  | otherwise = Node l              x (insert e r)
```

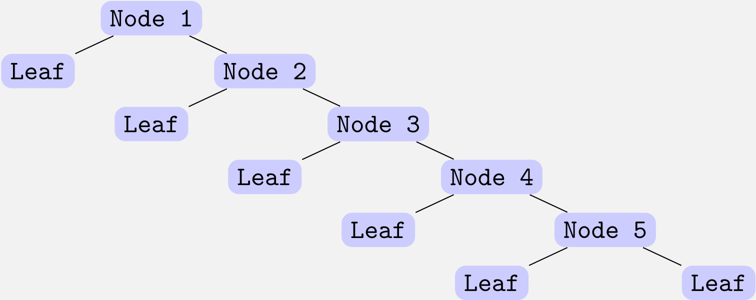
# `sort` for free!

1. Take a list `xs`
2. Build a search tree `toSearchTree xs`
    - The left-most element is the smallest
    - The right-most element is the largest
3. Turn it back into a list with `enumInfix`
4. The resulting list is sorted!

```
sort :: Ord a => [a] -> [a]
sort = enumInfix . toSearchTree
```

# Unbalanced search trees

```
> toSearchTree [1,2,3,4,5]
Node Leaf 1 (Node Leaf 2 ...))
```

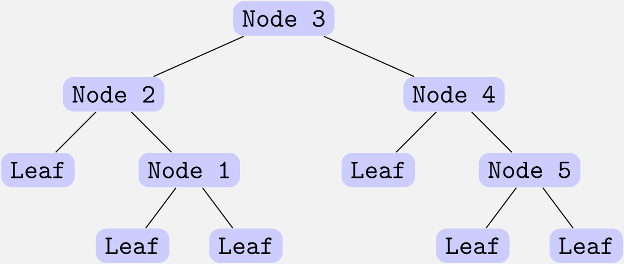



We win **nothing** by building this search tree

# Balanced search trees

Self-balancing trees keep their height at a minimum

- Close to the optimal minimum of $\log_2 n$
- 2-3 trees, red-black trees, AVL trees, …




*Reference: Purely Functional Data Structures* by Okasaki

# Delete from a search tree

`delete e t` returns the search tree `t` with `e` removed

- Respecting all the invariants from being a search tree

1. Define the type

```
delete :: Ord a => a -> Tree a -> Tree a
```

2. Enumerate the cases

```
delete e Leaf         = _
delete e (Node l x r) = _
```

3. Define the simple (base) cases
   - There is nothing to remove from an empty tree

```
delete e Leaf         = Leaf
```

# Delete from a search tree

4. Define the other (recursive) cases
   - We need to decide whether we have arrived to the node we want to remove

```
delete e (Node l x r)
  | e == x    = _  -- perform the deletion
  | e <  x    = Node (delete e l) x r
  | otherwise = Node l x (delete e r)
```

# Delete from a search tree

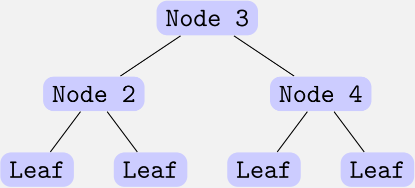
- When the data in the node is the one to remove, we are left with two search trees we need to turn into one
  1. If one of them is empty, we just take the other
     - `case` expr `of` performs further pattern matching
  2. In the other case, we need to find a new top value

```
delete e (Node l x r)
  | e == x = case (l, r) of
               (Leaf, _) -> r
               (_, Leaf) -> l
               _   -> let (x, l') = topValue l
                          in Node l' x r
```

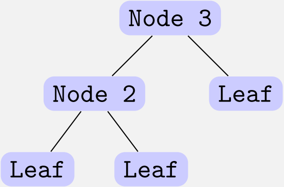
# Delete from a search tree

```
topValue :: Tree a -> (a, Tree a)
```

topValue




= 4 ,



# Delete from a search tree

In other words, `topValue t`

- Returns the right-most value in the tree
- Rebuilds the tree without it

```
topValue :: Tree a -> (a, Tree a)
topValue Leaf = error "no top value in empty tree"
topValue (Node _ x Leaf) = (x, l)
topValue (Node l x r)
  = let (y, r') = topValue r in (y, Node l x r')
```

# Delete from a search tree

In other words, `topValue t`

- Returns the right-most value in the tree
- Rebuilds the tree without it

```
topValue :: Tree a -> (a, Tree a)
topValue Leaf = error "no top value in empty tree"
topValue (Node _ x Leaf) = (x, l)
topValue (Node l x r)
  = let (y, r') = topValue r in (y, Node l x r')
```

There is a nice combinator for tuples, among others:

```
(<$>) :: (a -> b) -> (c, a) -> (c, b)
```

which allows us to rewrite the last line in a nicer way:

```
topValue (Node l x r) = Node l x <$> topValue r
```

# Key-value maps

# Key-value maps

A **map** keeps a list of present *keys* and associates a *value* with each one of them

```
lookup :: k -> Map k v -> Maybe v
```

We can define some extra functions using `lookup`

```
-- is the key present in the map?
member :: k -> Map k v -> Bool
member k m = isJust (lookup k m)

-- get the value or return a default one
findWithDefault :: v -> k -> Map k v -> v
findWithDefault def k m = case lookup k m of
                            Nothing -> def
                            Just v  -> v
```

# Association lists

A simple way to implement maps is to use a list of tuples

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

- `type` defines an **alias** or **type synonym**
  - Everytime we write `Map k v`, the compiler translates it to `[(k, v)]`
- Type synonyms are different from `data` declarations
  - `data` creates a *completely new* type
  - You need constructors to build or pattern match

# `lookup` for association lists

```
lookup :: Eq k => k -> Map k v -> Maybe v
lookup _ []     = Nothing
lookup e ((k,v) : rest)
  | e == k     = Just v
  | otherwise = lookup e rest
```

- The implementation follows the one for `elem`
- Suffers from the same bad characteristics
  - Linear cost for finding a key

# `lookup` for ordered association lists

If we guarantee that the keys are ordered, we can do better

```
lookup :: Ord k => k -> Map k v -> Maybe v
lookup _ []     = Nothing
lookup e ((k,v) : rest)
  | e == k     = Just v
  | e <  k     = Nothing
  | otherwise = lookup e rest
```

We can even go further and keep the map in a search tree

# `merge` for ordered association lists

`merge m1 m2` merges two given key-value maps:

- A key is present if it is present in any of both maps
- What should we do if the value is present in both maps?
  1. Choose arbitrarily the left or right element
  2. *Provide a way to configure the behavior*

1. Define the type

```
mergeWith :: Ord k
          => (v -> v -> v)   -- how to combine
          -> Map k v -> Map k v -> Map k v
```

# Cooking `merge`

2. Enumerate all the cases

```
mergeWith f [] [] = _
mergeWith f [] m2 = _
mergeWith f m1 [] = _
mergeWith f ((k1, v1) : r1) ((k2, v2) : r2)
                  = _
```

3. Define the simple (base) cases

```
mergeWith _ [] [] = []
mergeWith _ [] m2 = m2
mergeWith _ m1 [] = m1
```

# Cooking `merge`

4. Define the other (recursive) cases
   - We have to distinguish whether the key is the same
   - We need to output an ordered list

```
mergeWith f m1@((k1, v1) : r1) m2@((k2, v2) : r2)
  | k1 == k2 = (k1, f v1 v2) : mergeWith f r1 r2
  | k1 <  k2 = (k1, v1)      : mergeWith f r1 m2
  | k1 >  k2 = (k2, v2)      : mergeWith f m1 r2
```

# Merging with different bias

What should be the call to `mergeWith` to get?

- *Left bias*: prefer from the first argument
- *Right bias*: prefer from the second argument

## Questions

How do you define `f` for `mergeWith` to have those biases?

Is there any other notion which works well in this context?

# Merging with different bias

What should be the call to `mergeWith` to get?

- *Left bias*: prefer from the first argument
- *Right bias*: prefer from the second argument

## Questions

How do you define `f` for `mergeWith` to have those biases?

Is there any other notion which works well in this context?

```
mergeLeft  = mergeWith (\x _ -> x)
mergeRight = mergeWith (\_ y -> y)
```

# Monoids

Some types have an intrinsic notion of *combination*

- We already hinted at it when describing folds
- **Monoids** provide an *associative* binary operation with an *identity* element

```
class Monoid m where
  mempty  :: m
  mappend :: m -> m -> m
```

# Monoids

Some types have an intrinsic notion of *combination*

- We already hinted at it when describing folds
- **Monoids** provide an *associative* binary operation with an *identity* element

```
class Monoid m where
  mempty  :: m
  mappend :: m -> m -> m
```

Lists `[T]` are monoids *regardless* of their contained type `T`

```
instance Monoid [t] where
  mempty  = []    -- empty list
  mappend = (++)  -- concatenation
```

# Monoids as values

## `Monoid` provides sane defaults

```
lookup' :: (Ord k, Monoid v)
        => k -> Map k v -> v
lookup' = findWithDefault mempty

merge'  :: (Ord k, Monoid v)
        => Map k v -> Map k v -> Map k v
merge'  = mergeWith mappend
```

# Can we do better?

This is not part of the 2018/2019 course

- `lookup` and `merge` are expensive operations
  - We could enhance `lookup` with a search tree, but then `merge` becomes more expensive
- We impose at least an `Eq` constraint on the key

Nice but tricky code ahead!

# Inspiration: sets

A **set** of `T` is a data structure with operations

```
member :: t -> Set t -> Bool
union  :: Set t -> Set t -> Set t
```

Ordered lists provide a simple implementation

```
type Set t = [t]
member = elem
union  = merge
```

with all the disadvantages described for association lists

# Inspiration: sets

What if represent the set by its `member` function?

```
type Set t = t -> Bool

member :: t -> Set t        -> Bool
       -- t -> (t -> Bool) -> Bool
member e s = s e  -- apply the function
```

In mathematics, this representation is called an *indicator* or *characteristic* function for a set

Note that there is *no* `Eq` constraint over `t`

# Operations with indicator functions

```
union :: Set t         -> Set t         -> Set t
      -- (t -> Bool) -> (t -> Bool) -> t -> Bool
```

An element `e` is in the union of two sets `s1` and `s2` if it belongs to at least one of them

```
union s1 s2 = \e -> s1 e || s2 e
```

Intersection of sets is easy to define with indicator functions

# Operations with indicator functions

```
union :: Set t         -> Set t         -> Set t
      -- (t -> Bool) -> (t -> Bool) -> t -> Bool
```

An element `e` is in the union of two sets `s1` and `s2` if it belongs to at least one of them

```
union s1 s2 = \e -> s1 e || s2 e
```

Intersection of sets is easy to define with indicator functions

```
intersect :: Set t -> Set t -> Set t
intersect s1 s2 = \e -> s1 e && s2 e
```

# Key-value maps using functions

Let's apply the same idea and make maps equal to their `lookup` function

```
type Map k v = k -> Maybe v

lookup :: k -> Map k v -> Maybe v
lookup k m = m k
```

# mergeWith using functions

We look up the value in each of maps to be combined

- The only complex case is when the value is in both maps

```
mergeWith :: (v -> v -> v)
          -> Map k v -> Map k v -> Map k v
mergeWith f m1 m2
  = \k -> case (m1 k, m2 k) of
            (Nothing, v2) -> v2
            (v1, Nothing) -> v1
            (Just v1, Just v2) -> Just (f v1 v2)
```

# Left-biased `Maybe`

Haskell's standard library comes with a left-biased `Maybe`

```
data First a = First (Maybe a)

getFirst :: First a -> Maybe a
getFirst (First m) = m

instance Monoid (First a) where
  mempty = First Nothing
  mappend (First Nothing) y = y
  mappend x (First Nothing) = x
  mappend (First (Just x)) (First (Just _))
    = First (Just x)  -- prefer x over y
```

# Left-biased `merge`

We can exploit `First` monoid in our implementation

- We need to call `getFirst` in `lookup` to get a `Maybe v`
- Merging just combines the outcome of each map

```haskell
type Map k v = k -> First v

lookup :: k -> Map k v -> Maybe v
lookup k m = getFirst (m k)

merge :: Map k v -> Map k v -> Map k v
merge m1 m2 = \k -> m1 k `mappend` m2 k
```

# Left-biased `merge` with even less code

In the previous definition we exploit the instance

```
instance Monoid (First a) where ...
```

Actually, the library defines yet another `Monoid` instance

```
instance Monoid b => Monoid (a -> b) where ...
```

We can go one step further in reducing code

```
merge m1 m2 = m1 `mappend` m2
merge       = mappend  -- eta-reduction
```

# Disadvantages of functions

The implementation with functions is great, isn't it?

- It takes more memory if the map is big
- Everytime we ask for an element, we need to perform all the work
  - A lot if the maps were manipulated
  - Even when you intersect, the work becomes larger
- We cannot serialize a function easily
  - That is, transforming it to a format which we can write to disk or transmit via a network

# Summary

In this lecture we have practiced two important aspects

- Defining functions over trees by recursion
- Manipulate functions as data

We have also introduced the `Monoid` type class