Trees

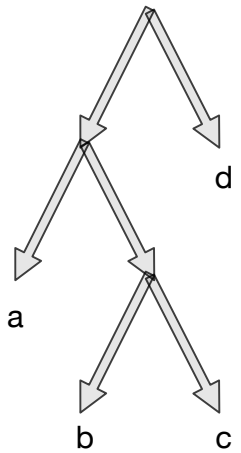
Professor William L. Harrison

September 26, 2016

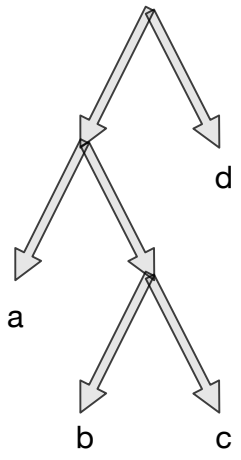
Today

- Starting Trees.
- Midterm on Monday, October 10th.

Binary Trees

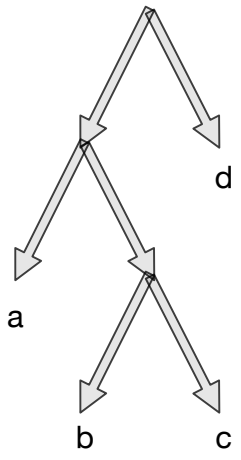


Binary Trees




```
data Btree a = Leaf a | Fork (Btree a) (Btree a)
```

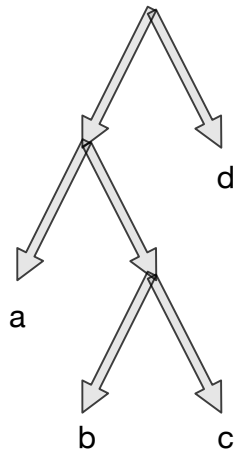
Binary Trees



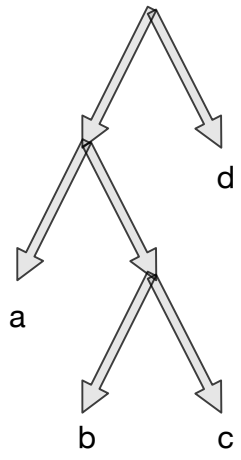

```
data Btree a = Leaf a | Fork (Btree a) (Btree a)
```

```
Fork (Fork (Leaf a) (Fork (Leaf b) (Leaf c))) (Leaf d)
```

Binary Trees



Binary Trees



- Leaf node is sometimes called an external node.
- Fork node is sometimes called an internal node.

Height

I.e., the length of the maximum path from the root to a leaf.

Q: What should the type of height be?

Height

I.e., the length of the maximum path from the root to a leaf.

Q: What should the type of height be?

```
height :: Btree a -> Int
height (Leaf _)      =
height (Fork t1 t2) =
```

Height

I.e., the length of the maximum path from the root to a leaf.

Q: What should the type of height be?

```
height :: Btree a -> Int
height (Leaf _)      = 0
height (Fork t1 t2) =
```

Height

I.e., the length of the maximum path from the root to a leaf.

Q: What should the type of height be?

```
height :: Btree a -> Int
height (Leaf _)      = 0
height (Fork t1 t2) = 1 + max (height t1) (height t2)
```

Size

I.e., How many leaves does it have?

Q: What should the type of size be?

Size

I.e., How many leaves does it have?

Q: What should the type of size be?

```
size :: Btree a -> Int
size (Leaf _)      =
size (Fork t1 t2) =
```

Size

I.e., How many leaves does it have?

Q: What should the type of size be?

```
size :: Btree a -> Int
size (Leaf _)      = 1
size (Fork t1 t2) =
```

Size

I.e., How many leaves does it have?

Q: What should the type of size be?

```
size :: Btree a -> Int
size (Leaf _)      = 1
size (Fork t1 t2) = size t1 + size t2
```

Flatten

I.e., the list with all the items at the leaves.

Q: What should the type of flatten be?

Flatten

I.e., the list with all the items at the leaves.

Q: What should the type of flatten be?

```
flatten :: Btree a -> [a]
flatten (Leaf x)     =
flatten (Fork t1 t2) =
```

Flatten

I.e., the list with all the items at the leaves.

Q: What should the type of flatten be?

```
flatten :: Btree a -> [a]
flatten (Leaf x)     = [x]
flatten (Fork t1 t2) =
```

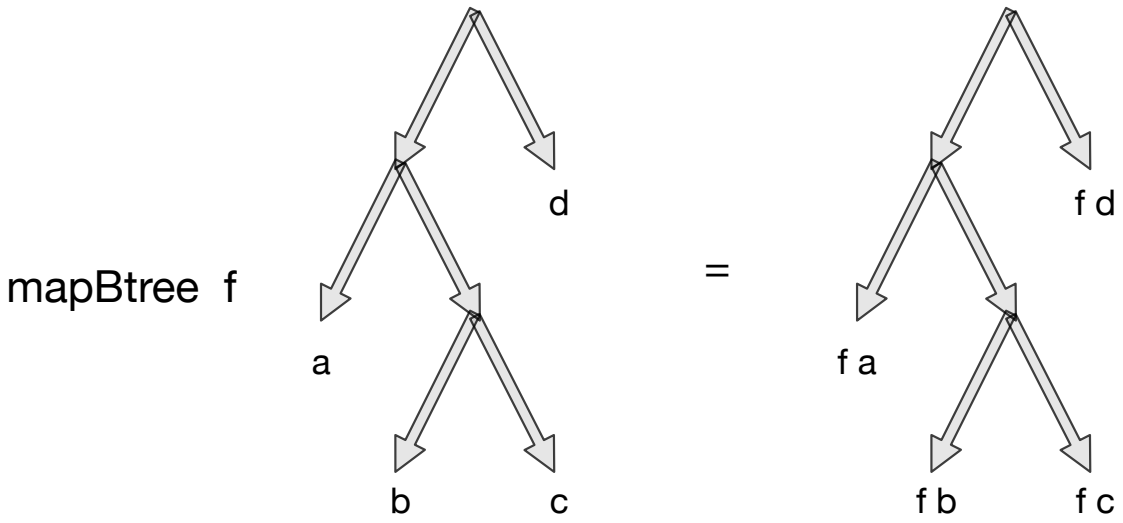
Flatten

I.e., the list with all the items at the leaves.

Q: What should the type of flatten be?

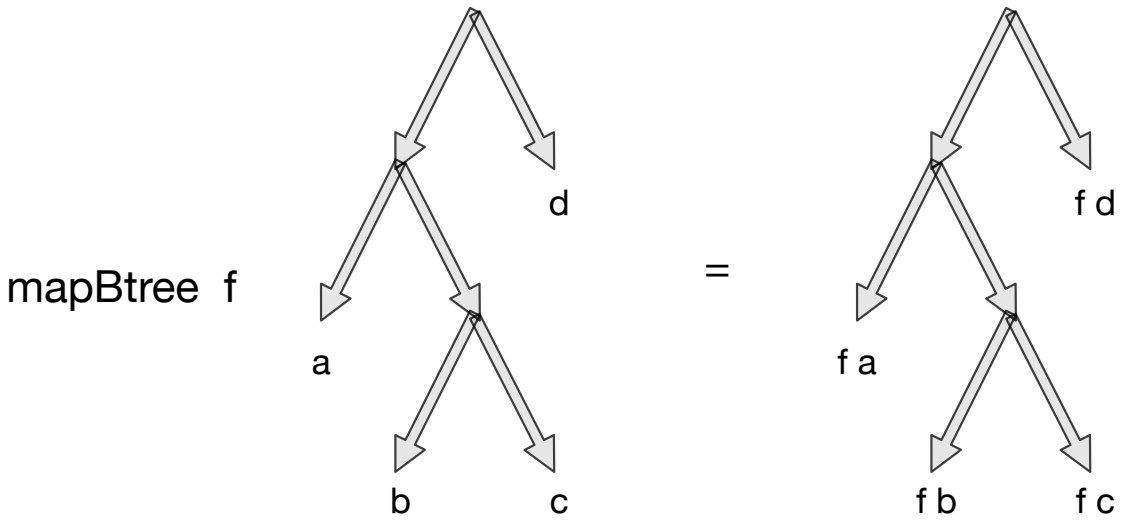
```
flatten :: Btree a -> [a]
flatten (Leaf x)      = [x]
flatten (Fork t1 t2) = flatten t1 ++ flatten t2
```

Map on Btree



Q: What should the type of mapBtree be?

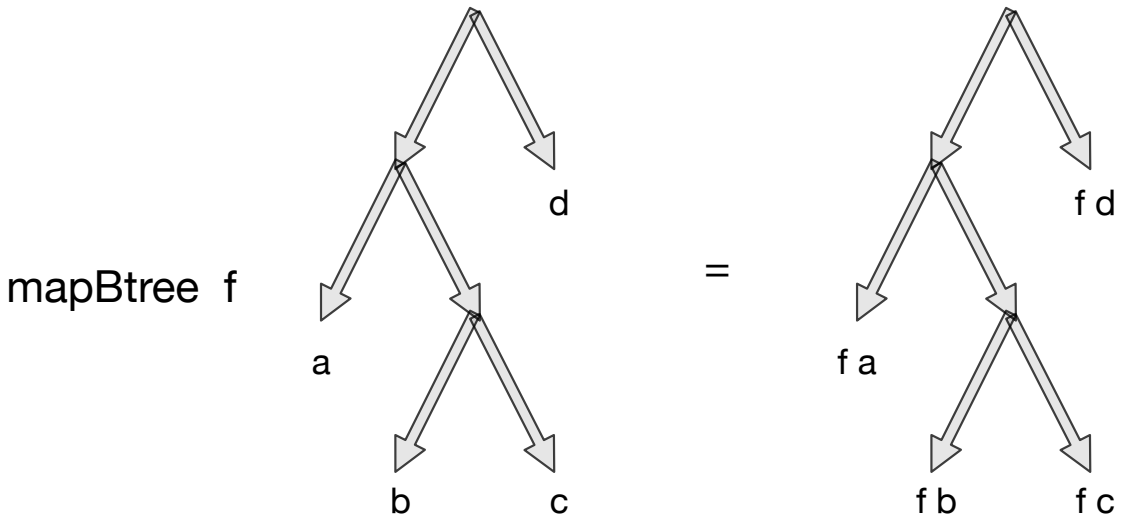
Map on Btree



Q: What should the type of mapBtree be?

```
mapBtree :: (a -> b) -> Btree a -> Btree b
mapBtree f (Leaf x)     =
mapBtree f (Fork t1 t2) =
```

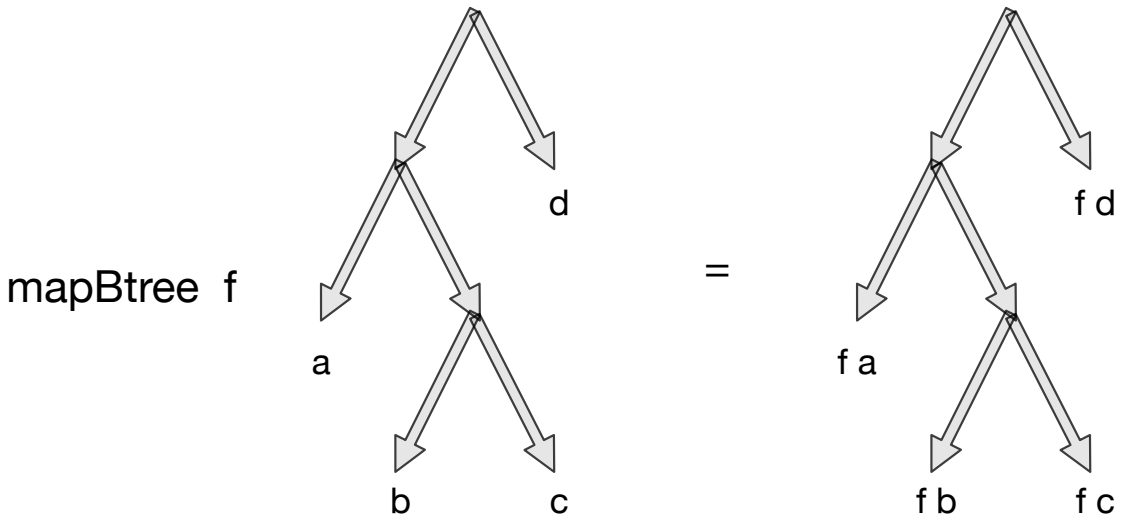
Map on Btree



Q: What should the type of mapBtree be?

```
mapBtree :: (a -> b) -> Btree a -> Btree b
mapBtree f (Leaf x)      = Leaf (f x)
mapBtree f (Fork t1 t2) =
```

Map on Btree



Q: What should the type of mapBtree be?

```
mapBtree :: (a -> b) -> Btree a -> Btree b
mapBtree f (Leaf x)      = Leaf (f x)
mapBtree f (Fork t1 t2) = Fork (mapBtree f t1) (mapBtree f t2)
```

Fold on Btree

Recall these definitions:

```
size :: Btree a -> Int
size (Leaf _)     = 1
size (Fork t1 t2) = size t1 + size t2

flatten :: Btree a -> [a]
flatten (Leaf x)     = [x]
flatten (Fork t1 t2) = flatten t1 ++ flatten t2
```

Fold on Btree

Recall these definitions:

```
size :: Btree a -> Int
size (Leaf _)     = 1
size (Fork t1 t2) = size t1 + size t2

flatten :: Btree a -> [a]
flatten (Leaf x)     = [x]
flatten (Fork t1 t2) = flatten t1 ++ flatten t2
```

Generalizing

```
foldBtree f g (Leaf x)     = f x
foldBtree f g (Fork t1 t2) =
          g (foldBtree f g t1) (foldBtree f g t2)
```

Fold on Btree (cont'd)

```
flatten :: Btree a -> [a]
flatten (Leaf x)      = [x]
flatten (Fork t1 t2) = flatten t1 ++ flatten t2

foldBtree f g (Leaf x)      = f x
foldBtree f g (Fork t1 t2) =
          g (foldBtree f g t1) (foldBtree f g t2)
```

Q: How do you write flatten as a call to foldBtree?

- f = ?
- g = ?

Fold on Btree (cont'd)

```
flatten :: Btree a -> [a]
flatten (Leaf x)      = [x]
flatten (Fork t1 t2) = flatten t1 ++ flatten t2

foldBtree f g (Leaf x)      = f x
foldBtree f g (Fork t1 t2) =
          g (foldBtree f g t1) (foldBtree f g t2)
```

Q: How do you write flatten as a call to foldBtree?

- f = \ x -> [x]
- g = ?

Fold on Btree (cont'd)

```
flatten :: Btree a -> [a]
flatten (Leaf x)      = [x]
flatten (Fork t1 t2) = flatten t1 ++ flatten t2

foldBtree f g (Leaf x)      = f x
foldBtree f g (Fork t1 t2) =
          g (foldBtree f g t1) (foldBtree f g t2)
```

Q: How do you write flatten as a call to foldBtree?

- f = \ x -> [x]
- g = (++)

Fold on Btree (cont'd)

```
flatten :: Btree a -> [a]
flatten (Leaf x)     = [x]
flatten (Fork t1 t2) = flatten t1 ++ flatten t2

foldBtree f g (Leaf x)     = f x
foldBtree f g (Fork t1 t2) =
          g (foldBtree f g t1) (foldBtree f g t2)
```

Q: How do you write flatten as a call to foldBtree?

- f = \ x -> [x]
- g = (++)
- flatten = foldBtree (\ x -> [x]) (++)

Fold on Btree (cont'd)

```
height :: Btree a -> Int
height (Leaf _)      = 0
height (Fork t1 t2) = 1 + max (height t1) (height t2)

foldBtree f g (Leaf x)      = f x
foldBtree f g (Fork t1 t2) =
          g (foldBtree f g t1) (foldBtree f g t2)
```

Q: How do you write height as a call to foldBtree?

- f = ?
- g = ?

Fold on Btree (cont'd)

```
height :: Btree a -> Int
height (Leaf _)      = 0
height (Fork t1 t2) = 1 + max (height t1) (height t2)

foldBtree f g (Leaf x)      = f x
foldBtree f g (Fork t1 t2) =
          g (foldBtree f g t1) (foldBtree f g t2)
```

Q: How do you write height as a call to foldBtree?

- f = \ _ -> 0
- g = ?

Fold on Btree (cont'd)

```
height :: Btree a -> Int
height (Leaf _)      = 0
height (Fork t1 t2) = 1 + max (height t1) (height t2)

foldBtree f g (Leaf x)      = f x
foldBtree f g (Fork t1 t2) =
          g (foldBtree f g t1) (foldBtree f g t2)
```

Q: How do you write height as a call to foldBtree?

- f = \ _ -> 0
- g = \ v1 v2 -> 1 + max v1 v2

Fold on Btree (cont'd)

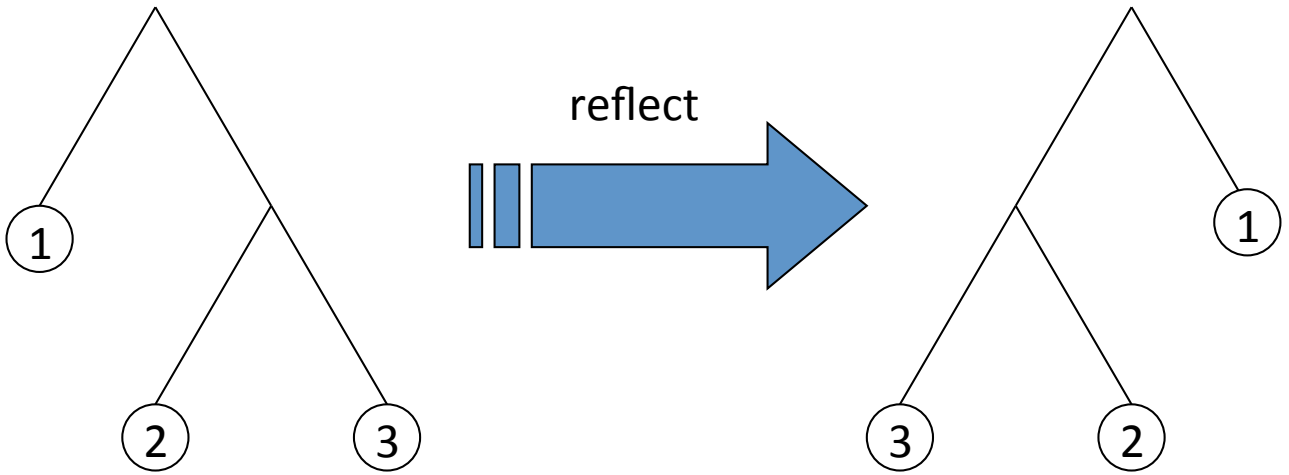
```
height :: Btree a -> Int
height (Leaf _)      = 0
height (Fork t1 t2) = 1 + max (height t1) (height t2)

foldBtree f g (Leaf x)      = f x
foldBtree f g (Fork t1 t2) =
          g (foldBtree f g t1) (foldBtree f g t2)
```

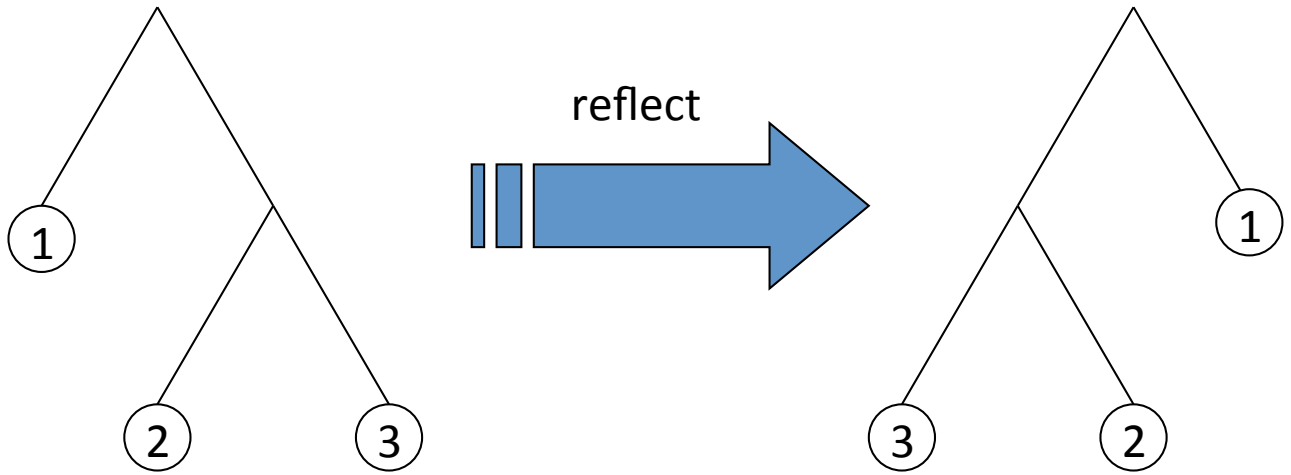
Q: How do you write height as a call to foldBtree?

- f = \ _ -> 0
- g = \ v1 v2 -> 1 + max v1 v2
- height = foldBtree (_ -> 0) (\ v1 v2 -> 1 + max v1 v2)

Reflection

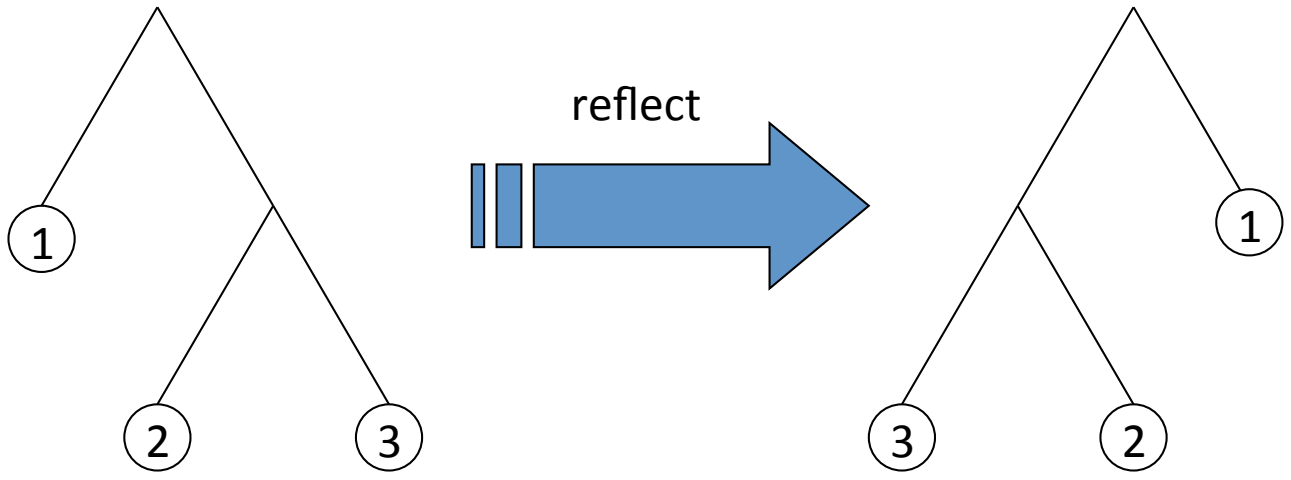


Reflection



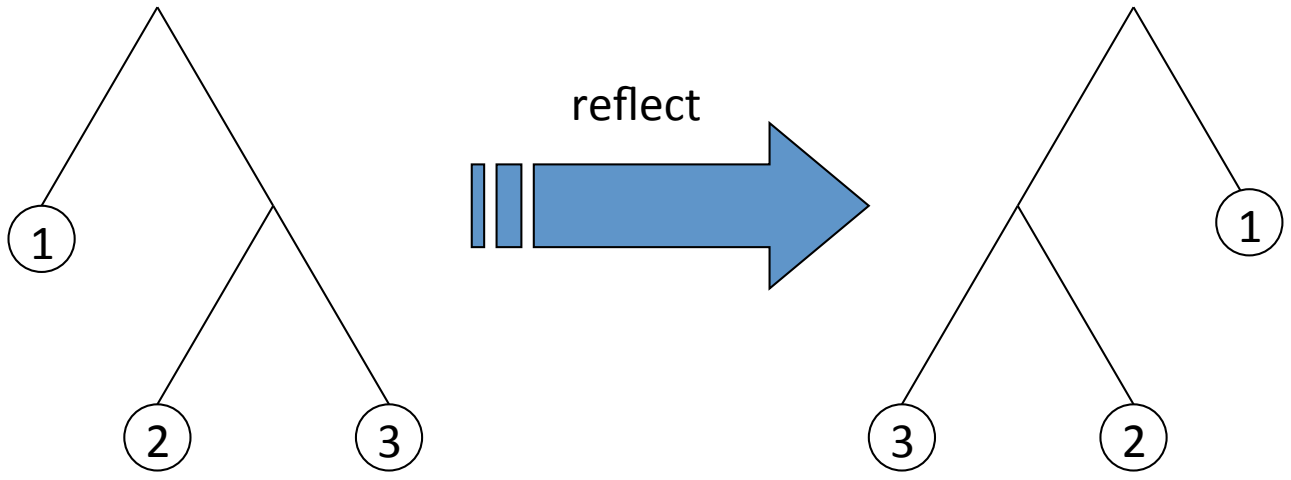

```
reflect :: Btree a -> Btree a
reflect (Leaf a)     = ?
reflect (Fork t1 t2) = ?
```

Reflection



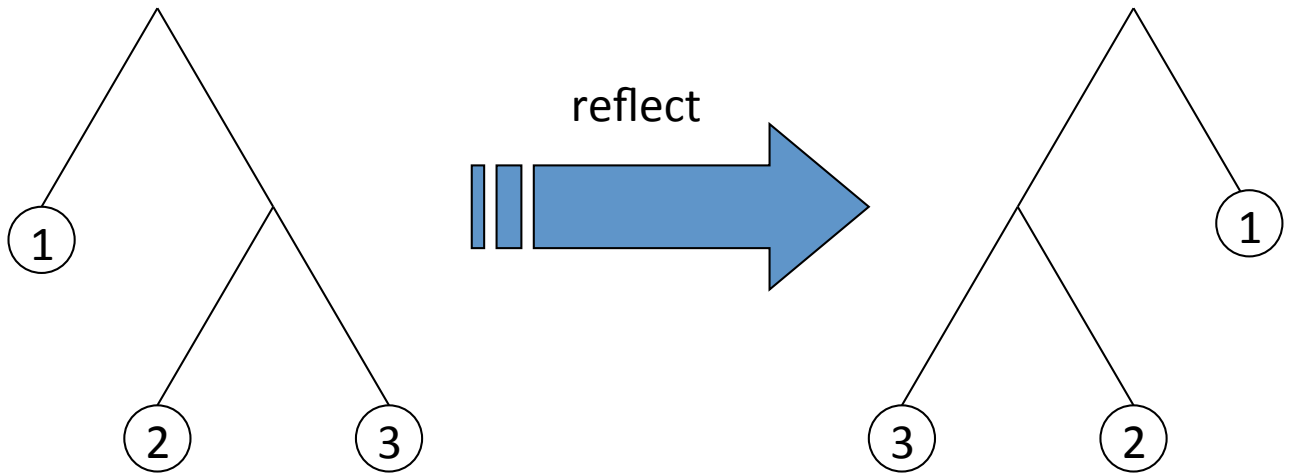

```
reflect :: Btree a -> Btree a
reflect (Leaf a)      = Leaf a
reflect (Fork t1 t2) = ?
```

Reflection




```
reflect :: Btree a -> Btree a
reflect (Leaf a)      = Leaf a
reflect (Fork t1 t2) = Fork (reflect t2) (reflect t1)
```

Reflection




```
reflect :: Btree a -> Btree a
reflect (Leaf a)      = Leaf a
reflect (Fork t1 t2) = Fork (reflect t2) (reflect t1)
```

Q: Is reflect a tree fold?