Lecture 7

```
qsort' :: Ord a => [a] -> [a]
qsort' [] = []
qsort' (x:xs) = qsort' ys ++ [x] ++ qsort' zs
  where
    (ys, zs) = partition (<= x) xs

partition :: (a -> Bool) -> [a] -> ([a], [a])
partition p xs = (filter p xs, filter (not . p) xs)

(<= x) :: Ord a => a -> Bool.
(<= x) y = y <= x
```

```
data Maybe a = Nothing
             | Just a
```

{-# LANGUAGE GADTs #-}

```
data Maybe a where
  Nothing :: Maybe a
  Just    :: a -> Maybe a
```

```
fromMaybe :: Maybe a → [a]
fromMaybe  Nothing  = []
fromMaybe  (Just x) = [x]
```

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

```
data Tree a where
    Tip  :: Tree a
    Node :: Tree a → a → Tree a → Tree a
```

qsort [5, 3, 1, 4, 2, 6]

=

flatten (mkTree [5, 3, 1, 4, 2, 6])

=

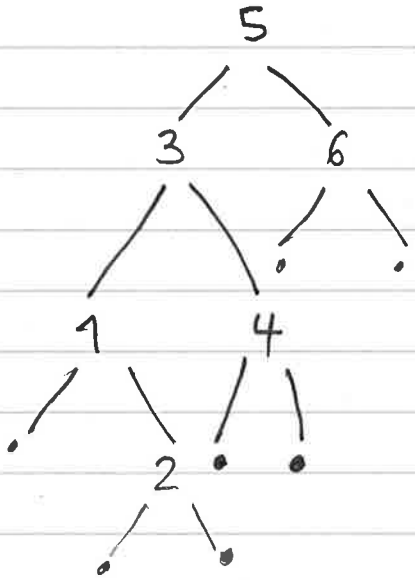
flatten (Node (mkTree [3, 1, 4, 2])
5
(mkTree [6]))

mkTree [3, 1, 4, 2]
= Node (mkTree [1, 2])
3
(mkTree [4])

mkTree [1, 2]
= Node (mkTree [])
1
(mkTree [2])

mkTree [] = Tip

mkTree [2] = Node (mkTree [])
2
(mkTree [])


5
3
6
1
4
2

Functors.

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

Maybe a
Tree a
[a] = [] a
List a

```
instance Functor [] where
    fmap :: (a -> b) -> [a] -> [b]
    fmap = map

instance Functor Maybe where
    fmap :: (a -> b) -> Maybe a -> Maybe b
    fmap f Nothing = Nothing
    fmap f (Just x) = Just (f x)
```

f : a -> b
x : a

Example.

```
Nothing :: Maybe Int
alpha :: Int -> Char
alpha 1 = 'a'
  :
alpha 26 = 'z'
```

```
fmap alpha Nothing =
    Nothing
fmap alpha (Just 2) =
    Just 'b'.
```

fmap alpha [1,2,3] = ['a','b','c']

instance Functor Tree where

~~fmap :: (a→b) → f a → f b~~

fmap :: (a→b) → Tree a → Tree b

fmap f Tip = Tip

fmap f (Node l x r) =

f : a→b, l : Tree a, x : a, r : Tree a, Tree b

Node (fmap f l)
(f x) ← b
(fmap f r)
← Tree b

fmap alpha