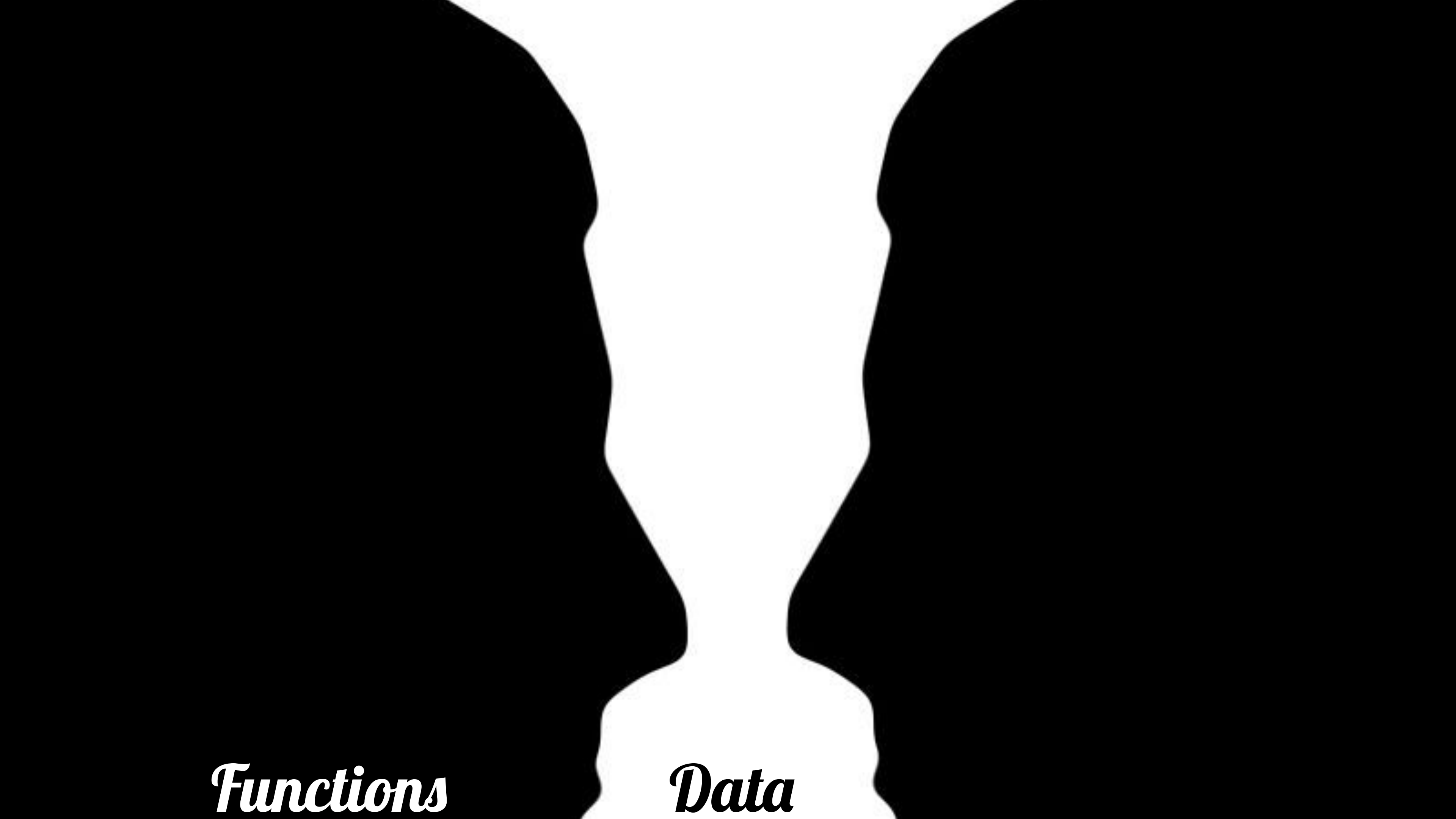

Functions
Data

Today’s Superpower

Interweaving functions and data structures for much goodness.

Quick Recap

```haskell
double [] = []
double (x:xs) = 2*x : double fn xs

Prelude> double [ 1, 2, 3 ]
```

```haskell
sumList [] = 0
sumList (x:xs) = x + sumList xs
Prelude> sumList [ 1, 2, 3 ]
```

```
[ 1,    2,    3 ]
  ↓     ↓     ↓
 (*2)  (*2)  (*2)
  ↓     ↓     ↓
[ 2,    4,    6 ]
```

```
sumList [ 1, 2, 3 ]

sumList 1 : 2 : 3 : []

1 + sumList 2 : 3 : []
1 + 2 + sumList 3 : []
1 + 2 + 3 + sumList []
1 + 2 + 3 + 0
6
```

```haskell
double [] = []
double (x:xs) = 2*x : double xs

wordLen [] = []
wordLen (x:xs) = length x : wordLen xs

rem7 [] = []
rem7 (x:xs) = rem x 7 : rem7 xs
```

```haskell
double [1,2,3]                  => [2,4,6]

wordLen [“bee”,”deer”,”egret”]  => [3,4,5]

rem7 [6,7,8]                    => [6,0,1]
```

```
double [] = []
double (x:xs)   = 2*x : double xs

wordLen [] = []
wordLen (x:xs)  = length x : wordLen xs

rem7 [] = []
rem7 (x:xs)      = rem x 7 : rem7 xs
```

Three different functions

But not that different-what two things differentiates each from the others?

~~Function name~~

~~`double`, `wordLen`, `rem7`~~

The function applied to each element

(*2) , length, (flip rem 7)

```haskell
double [] = []
double (x:xs)   = 2*x : double xs

wordLen [] = []
wordLen (x:xs)  = length x : wordLen xs

rem7 [] = []
rem7 (x:xs)      = rem x 7 : rem7 xs
```

Three different functions

What differentiates each from the others?

The function applied to each element

(*2) , length, (flip rem 7)

```haskell
mapper _fn []     = []

mapper  fn (x:xs) = fn x : mapper fn xs
```

```
*Main> mapper (*4) [1,2,3]
[4,8,12]

*Main> mapper length ["ant","bear"]
[3,4]

*Main> mapper (flip rem 7) [6,7,8]
[6,0,1]
```

```
mapper _fn []     = []

mapper  fn (x:xs) = fn x : mapper fn xs
```

Standard Haskell library function: **map**

```
*Main> map (*4) [1,2,3]
[4,8,12]

*Main> map length ["ant","bear"]
[3,4]

*Main> map (flip rem 7) [6,7,8]
[6,0,1]
```

```
double [] = []
double (x:xs) = 2*x : double fn xs

Prelude> double [ 1, 2, 3 ]
```

```
[ 1,    2,    3 ]
  ↓     ↓     ↓
 (*2)  (*2)  (*2)
  ↓           ↓
[ 2,
```

```
sumList [] = 0
sumList (x:xs) = x + sumList xs
Prelude> sumList [ 1, 2, 3 ]
```

```
sumList [ 1, 2, 3 ]

sumList 1 : 2 : 3 : []

1 + sumList 2 : 3 : []
1 + 2 + sumList 3 : []
1 + 2 + 3 + sumList []
1 + 2 + 3 + 0
6
```

```
sumList [] = 0
sumList (x:xs) = x + sumList xs

timesList [] = 1
timesList (x:xs) = x * timesList xs

len [] = 0
len (x:xs) = 1 + len xs
```

What two things differentiates each function from the others?

The function applied to each element

(x+y) , (x*y), (1+y)

The initial (empty list) value

0, 1, 0

```
reduce _fn value [] = value
reduce  fn value (x:xs) = reduce fn (fn value x) xs
```

```
reduce _fn value [] = value
reduce  fn value (x:xs) = reduce fn (fn value x) xs
```

```
reduce (+) 0 [1,2,3]
reduce (+) (value + x) [2, 3]
reduce (+) (0 + 1) [2, 3]
reduce (+) (1 + 2) [3]
reduce (+) (3 + 3) []
6
```

Haskell calls this function a *fold*.

Because it consumes data from the left, its name is **`foldl`**

Map

```
double [] = []
double (x:xs) = 2*x : double fn xs

Prelude> double = map (*2)
```

Use map to transform a list into another list by applying a function to each element in turn.

Fold

```
sumList [] = 0
sumList (x:xs) = x + sumList xs

Prelude> sumList = foldl (+) 0
```

Use fold to reduce a list into a single value.

(The single value might itself be a list...)

Quick lab...

Let’s play with map and foldl using GHCI.

- Use map to convert a list of strings to a list of those strings’ lengths.
- A convenient way to see if two words are anagrams is to sort the letters in each. If the sorted values are the same, then the two words contain the same letters, and are anagrams. Write a function that takes a list of words and returns a list of each word sorted. (So [“ant”, “cat”, “tan”] would return [“ant”, “act”, “ant”])
- Use reduce the find the largest value in a list of positive numbers. The largest value in an empty list is zero.
- Use reduce to concatenate the values in a list of strings (the ++ operator merges two strings)

```
double [] = []
double (x:xs) = 2*x : double fn xs

Prelude> double = map (*2)
```

```
sumList [] = 0
sumList (x:xs) = x + sumList xs

Prelude> sumList = foldl (+) 0
```

Defining Data Types

Switch:

In code:

```
// Boolean
switch = true
switch = false
```

Switch:

On

Off

In code:

```
// String
switch = “closed”
switch = “open”
```

Switch:

On

Off

In code:

```
// Integer
switch = 1
switch = 0
```

Switch: **On** **Off**

In code:

```
// Object
switch = new Switch()
switch.close()
switch.open()
```

All I wanted was a variable that can just hold two values!

Switch: On Off

In Haskell:

```
data Switch = SwitchOn | SwitchOff

switch = SwitchOn
other
```

```haskell
data Switch = SwitchOn | SwitchOff

sw1 = SwitchOn
sw2 = SwitchOff
```

```haskell
data Switch = SwitchOn | SwitchOff

sw1 = SwitchOn
sw2 = SwitchOff

toggle SwitchOn  = SwitchOff
toggle SwitchOff = SwitchOn

sw1’ = toggle sw1

roomLights = [ SwitchOn, SwitchOn, SwitchOff ]

newScene = map toggle roomLights
```

```haskell
data Switch = SwitchOn | SwitchOff

sw1 = SwitchOn
sw2 = SwitchOff

toggle SwitchOn  = SwitchOff
toggle SwitchOff = SwitchOn

sw1’ = toggle sw1

roomLights = [ SwitchOn, SwitchOn, SwitchOff ]

newScene = map toggle roomLights
```

All you can do with type Switch is assign it and pattern match against it. You can’t even do `show sw1`

```haskell
data Switch = SwitchOn | SwitchOff
  Deriving (Show, Eq)

sw1 = SwitchOn
sw2 = SwitchOff

toggle SwitchOn  = SwitchOff
toggle SwitchOff = SwitchOn

sw1’ = toggle sw1

roomLights = [ SwitchOn, SwitchOn, SwitchOff ]

newScene = map toggle roomLights
```

Deriving tells Haskell to apply some defaults for common capabilities (such as showing, equality, and …)

100%

:

50%

:

Off

In Haskell:

```haskell
data Dimmer = DimmerOff |
              DimmerOn Float
```

100%

:

50%

:

Off

In Haskell:

```haskell
data Dimmer = DimmerOff |
              DimmerOn Float


dim1 = DimmerOn 50
dim2 = DimmerOn 80
dim3 = DimmerOff
```

In Haskell:

```haskell
data Dimmer = DimmerOff |
              DimmerOn Float

dim1 = DimmerOn 50
dim2 = DimmerOn 80
dim3 = DimmerOff

isBright DimmerOff = False
isBright (DimmerOn pc) = pc >= 75
```

100%

:

50%

:

Off

In Haskell:

```haskell
data Dimmer = DimmerOff |
              DimmerOn Float

dim1 = DimmerOn 50
dim2 = DimmerOn 80
dim3 = DimmerOff

isBright DimmerOff = False
isBright (DimmerOn pc) = pc >= 75

isBright dim1    -- False
isBright dim2    -- True
isBright dim3    -- False
```

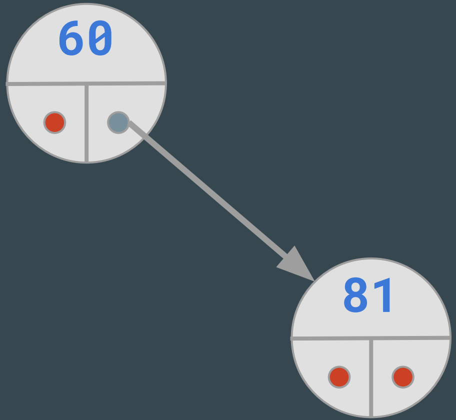

I think that I shall never see
A poem lovely as a tree.
Joyce Kilmer

tree = Empty

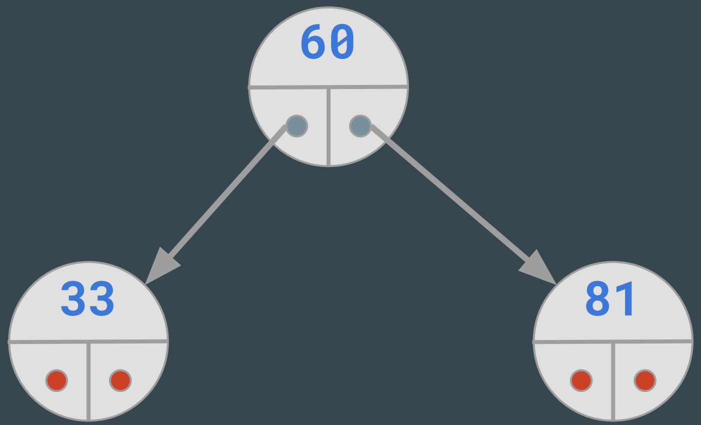
tree = Empty

Add 60

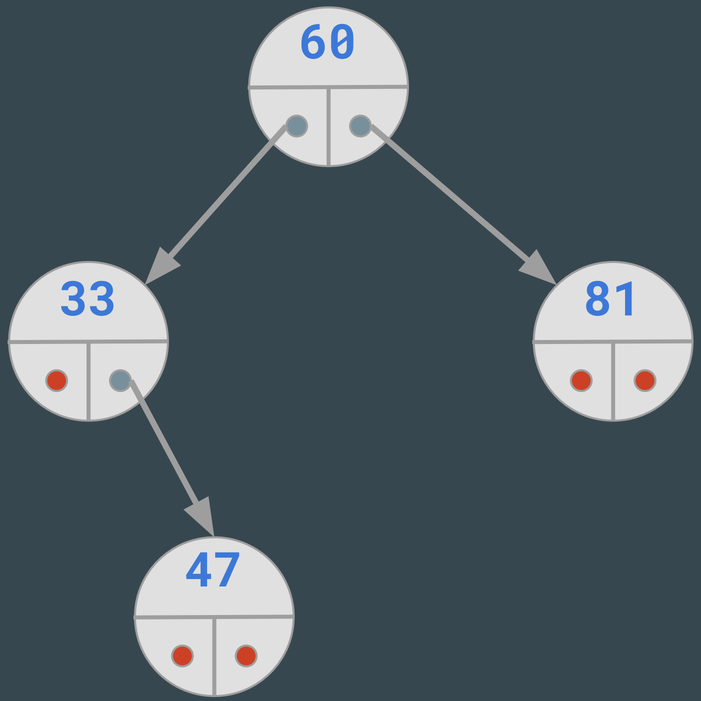
tree = Empty
Add 60
Add 81



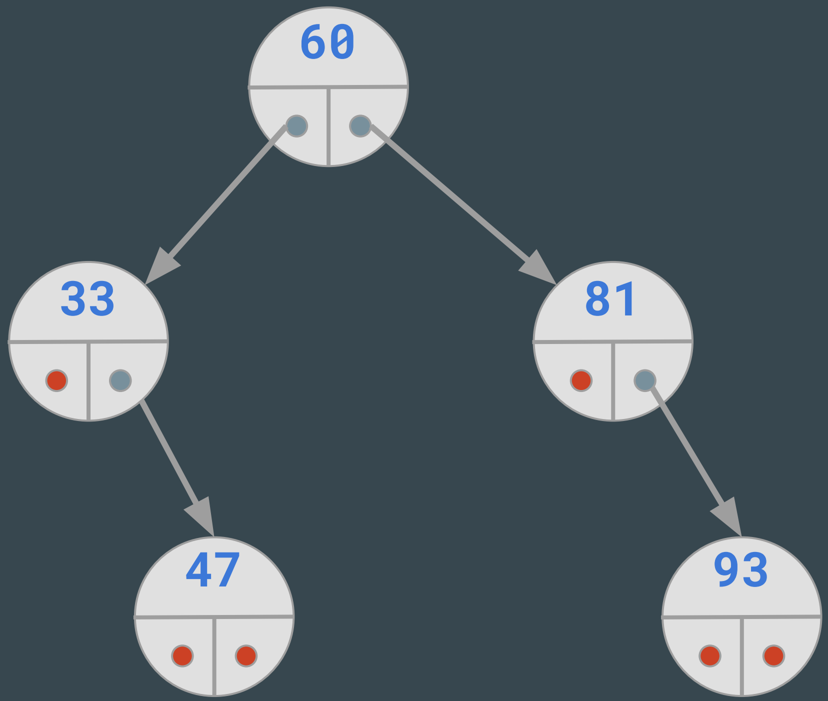
tree = Empty
Add 60
Add 81
Add 33



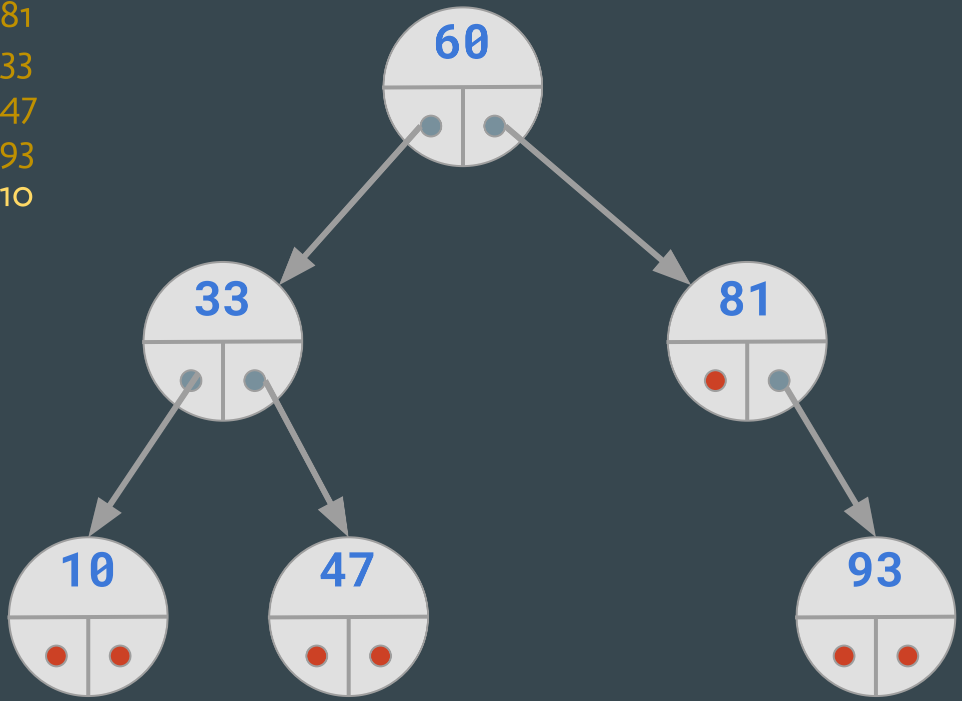
tree = Empty
Add 60
Add 81
Add 33
Add 47

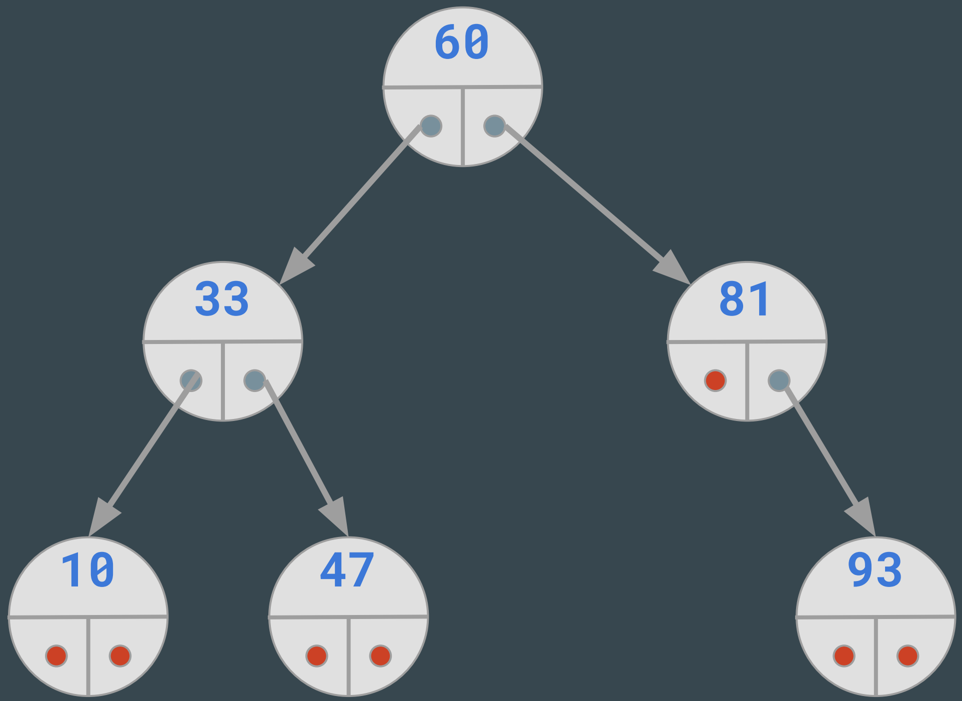


tree = Empty
Add 60
Add 81
Add 33
Add 47
Add 93



tree = Empty
Add 60
Add 81
Add 33
Add 47
Add 93
Add 10



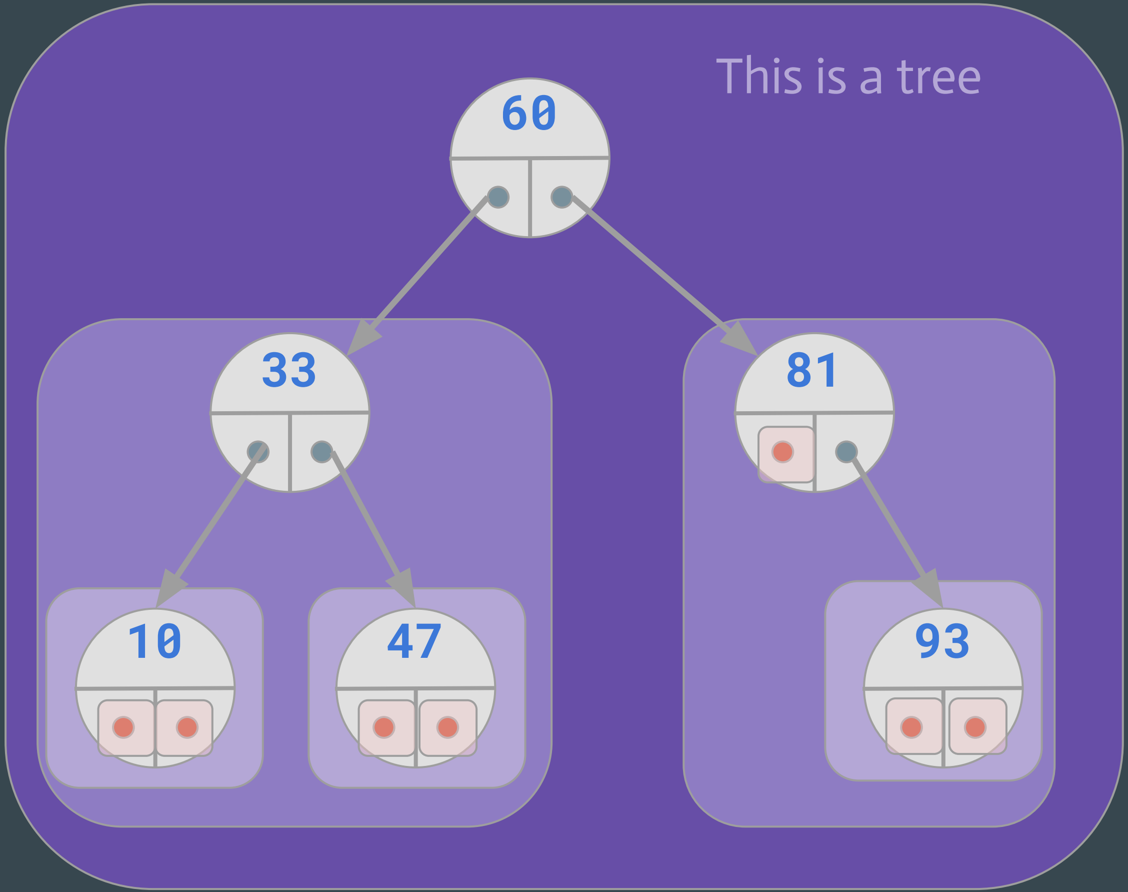


A Tree is:

Value:	(integer)
Left child:	Tree
Right child:	Tree

-or-

Empty



A Tree is:

Value:	(integer)
Left child:	Tree
Right child:	Tree

-or-

Empty

```haskell
data Tree = Empty
          | Branch Tree Tree Integer
          deriving (Show, Eq)
```

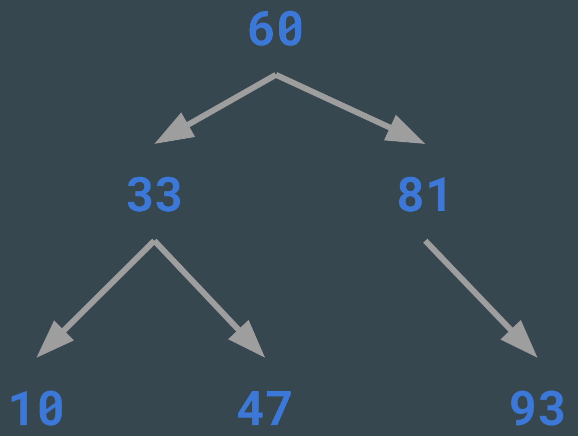
A Tree is:

Value:	(integer)
Left child:	Tree
Right child:	Tree

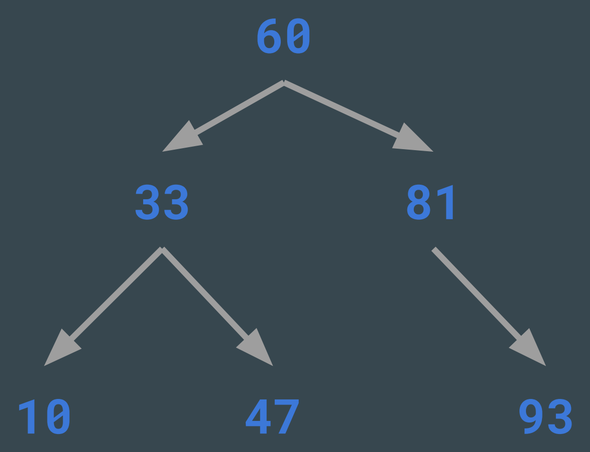
-or-

Empty

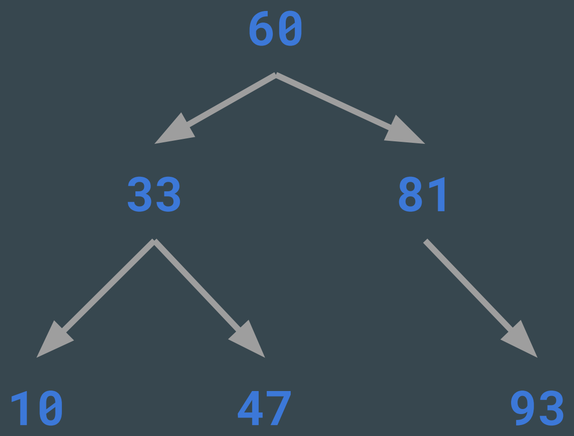
```haskell
data Tree = Empty
          | Branch Tree Tree Integer
          deriving (Show, Eq)

Branch
   (Branch
         (Branch Empty Empty 10)
         (Branch Empty Empty 47)
         33
   )
   (Branch
         Empty
         (Branch Empty Empty 93)
         81
   )
   60
```



```haskell
data Tree = Empty
          | Branch Tree Tree Integer
          deriving (Show, Eq)

tsum Empty = 0
tsum (Branch left right value) =
    value + tsum left + tsum right
```



```haskell
data Tree = Empty
          | Branch Tree Tree Integer
          deriving (Show, Eq)

tfold fn initialValue Empty = initialValue
tfold fn initialValue (Branch l r v) =
   fn v (fn foldLeftChild foldRightChild)
   where
       foldLeftChild  = doFold l
       foldRightChild = doFold r
       doFold = tfold fn initialValue

tsum :: Tree -> Integer
tsum = tfold (+) 0
```



```haskell
tadd :: Tree -> Integer -> Tree
tadd Empty value = Branch Empty Empty value
tadd (Branch l r nodeValue) newValue
 | newValue < nodeValue = Branch (tadd l newValue) r nodeValue
 | otherwise            = Branch l (tadd r newValue) nodeValue

tbuild:: [Integer] -> Tree
tbuild = foldl tadd Empty
```

```
Prelude> tbuild [6,5,9]
Branch (Branch Empty Empty 5) (Branch Empty Empty 9) 6
```

Homework...

Due Tuesday, May 11

Fork and clone the repo https://github.com/Eastside-FP/dayfive You’ll be working on code the the assignment/ directory, and submitting using a pull request.

Fork and clone the repo https://github.com/Eastside-FP/dayfive. You’ll be working on code the the assignment/ directory, and submitting using a pull request.

In that directory, you’ll find the file assignment.hs.

It contains placeholders for code that you’ll write, along with a set of tests.

Update the placeholders to make them do what the description says, and so that the tests pass.

Due Tuesday, May 11