Programming in Haskell
Aug-Nov 2015

LECTURE 16

OCTOBER 13, 2015

S P Suresh
Chennai Mathematical Institute

Recursive data types

- Just like we have recursive functions, we can have recursive data types
- A recursive datatype **T** is one which has some components of the same type **T**
- Some constructors of a recursive data type **T** have **T** among the input types, as well as the return type

First example: Nat

- Simplest example is `Nat`
- `data Nat = Zero | Succ Nat`
- `Zero :: Nat`
- `Succ :: Nat -> Nat`

Nat

- ```haskell
  iszero :: Nat -> Bool
  iszero Zero     = True
  iszero (Succ _) = False
  ```

- ```haskell
  pred :: Nat -> Nat
  pred Zero       = Zero
  pred (Succ n)   = n
  ```

Nat

* ```
  plus :: Nat -> Nat -> Nat
  plus m Zero     = m
  plus m (Succ n) = Succ (plus m n)
  ```

* ```
  mult :: Nat -> Nat -> Nat
  mult m Zero     = Zero
  mult m (Succ n) = plus ((mult m n) m)
  ```

Second example: List

- Recursive data types can also be polymorphic
- `List a = Nil | Cons a (List a)`
- This is the built-in type `[a]`

List

- Functions are defined as usual using pattern matching

- ```
  head :: List a -> a
  head (Cons x _) = x
  ```

- This causes an exception on `head Nil`

- You can have your preferred behaviour

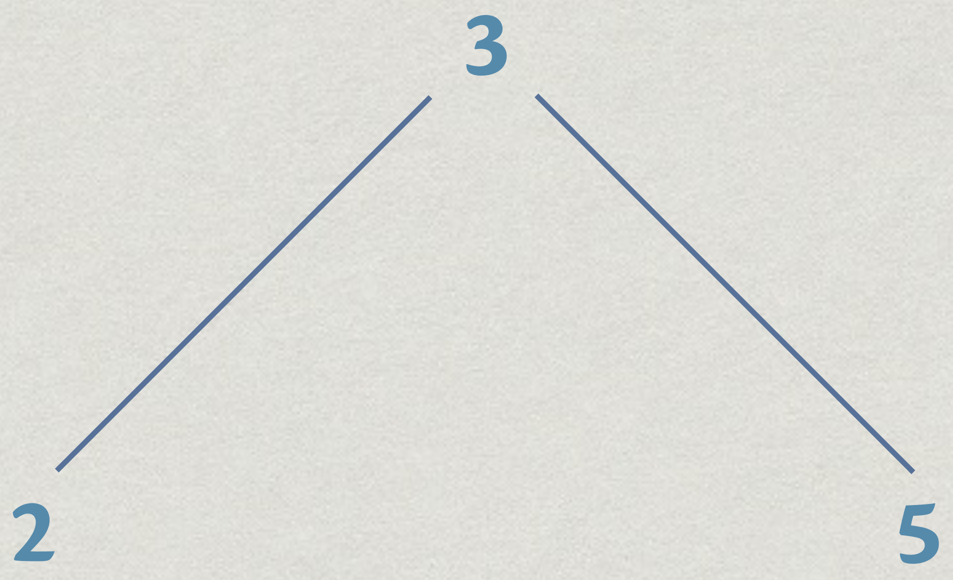
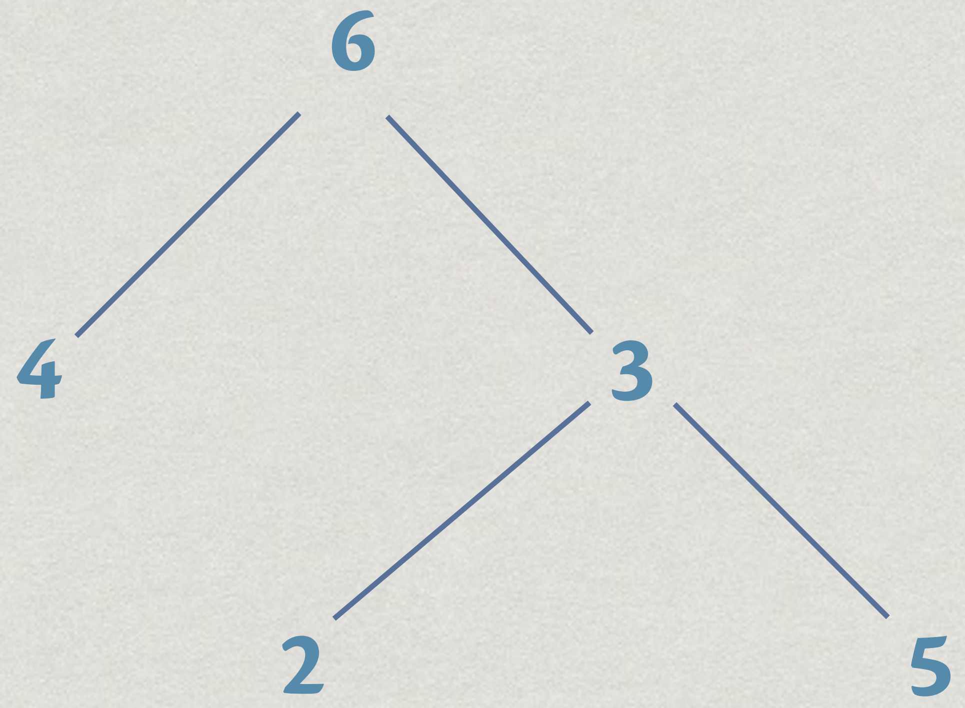
- ```
  head :: List a -> Maybe a
  head Nil          = Nothing
  head (Cons x _) = Just x
  ```

Binary trees

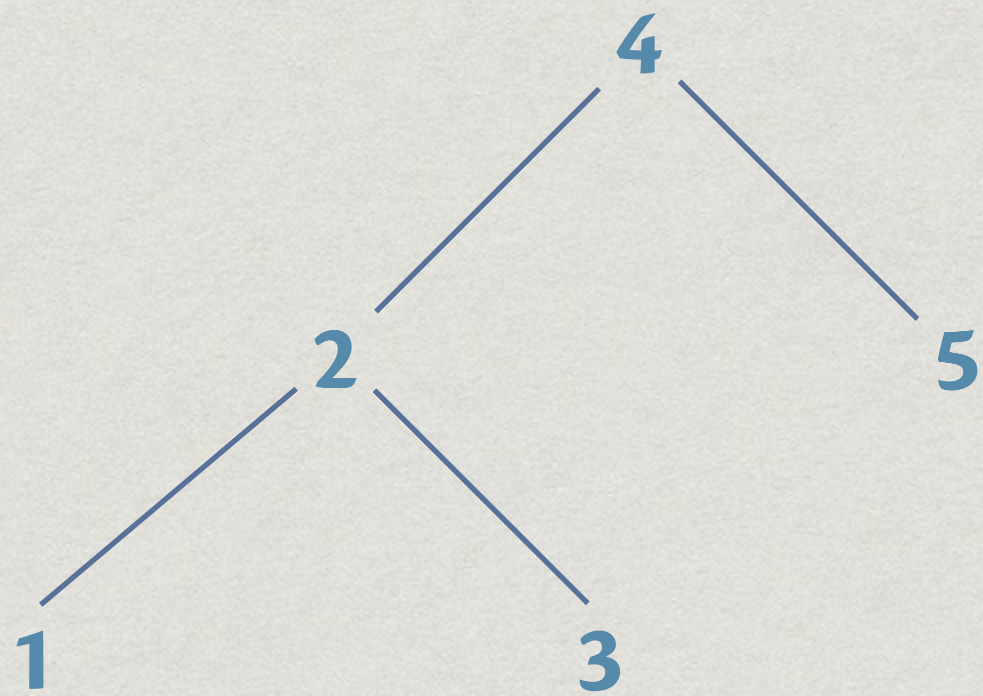
- A binary tree data structure is defined as follows:
  - The empty tree is a binary tree
  - A node containing an element with left and right subtrees is a binary tree
- ```
  data BTree a = Nil
               | Node (BTree a) a (BTree a)
  ```
  ```

Binary trees

- ```
  Nil  :: BTree a
  Node :: BTree a -> a -> BTree a -> BTree a
  ```

- ```
  Node (Node Nil 2 Nil) 3
             (Node Nil 5 Nil)
  ```

- ```
  Node (Node Nil 4 Nil) 6
      (Node (Node Nil 2 Nil) 3
                 (Node Nil 5 Nil))
  ```
  ```

Binary trees

- ```
  Node (Node Nil 2 Nil) 3
              (Node Nil 5 Nil)
  ```

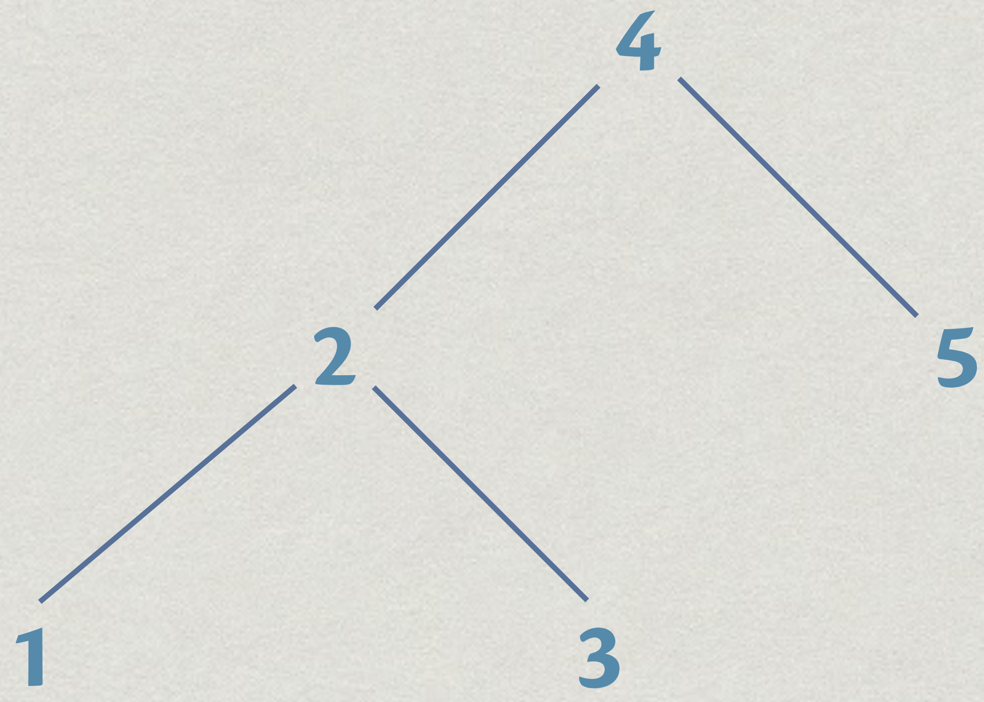
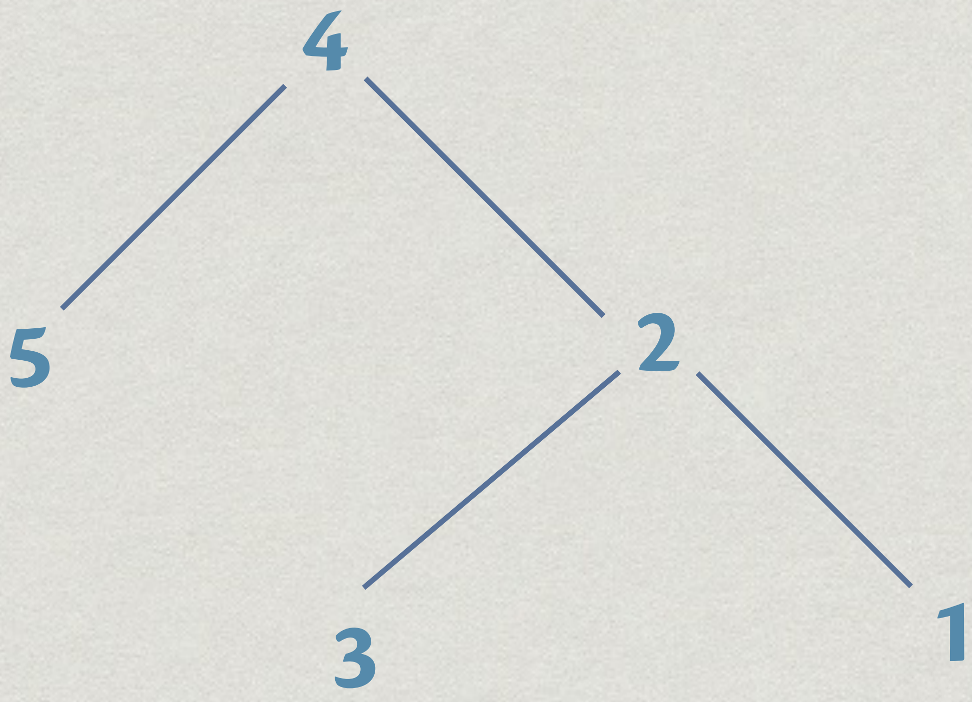

  ```

Binary trees

- ```
  Node (Node Nil 4 Nil) 6
       (Node (Node Nil 2 Nil) 3
                    (Node Nil 5 Nil))
  ```


  ```

Binary trees

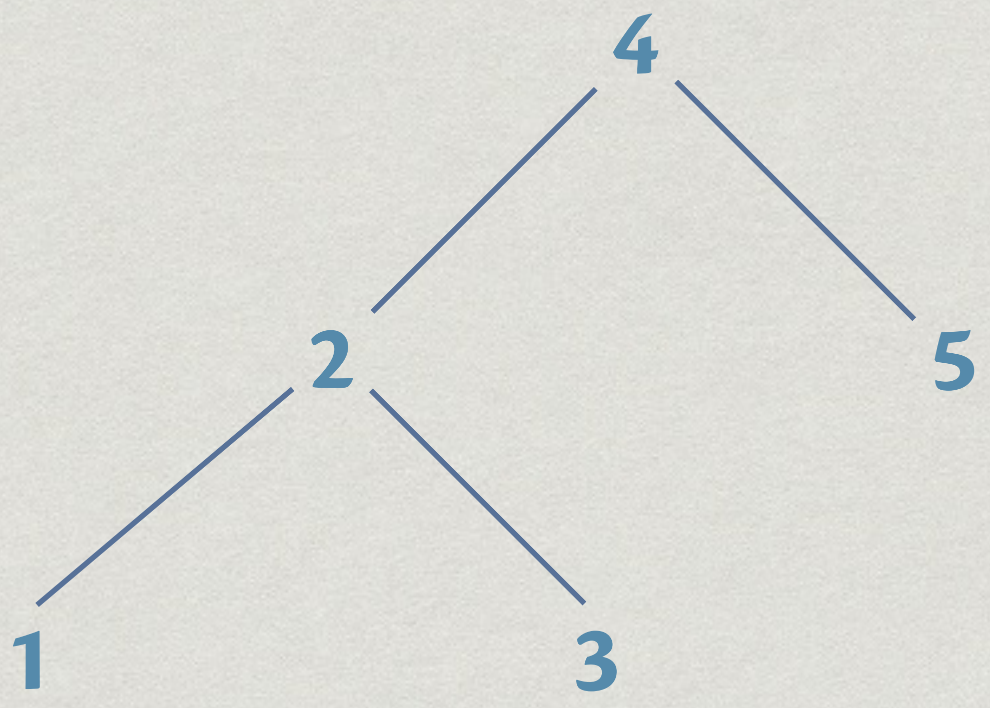



- ```
  Node  (Node (Node Nil 1 Nil) 2
                   (Node Nil 3 Nil))
      4
        (Node Nil 5 Nil)
  ```
  ```

Functions on binary trees

- `size` - Number of nodes in a tree
- ```
  size :: BTree a -> Int
  size Nil                = 0
  size (Node tl x tr) = size tl + 1 + size tr
  ```
  ```

Functions on binary trees

- `height` - Longest path from root to leaf
- ```
  height :: BTree a -> Int
  height Nil               = 0
  height (Node tl x tr) = 1 +
                          max (height tl) (height tr)
  ```
  ```

Functions on binary trees

reflect - Reflect the tree on the “vertical axis”





Functions on binary trees

* `reflect` - Reflect the tree on the "vertical axis"
* ```
  reflect :: BTree a -> BTree a
  reflect Nil                 = Nil
  reflect (Node tl x tr) = Node
                               (reflect tr)
                                    x
                               (reflect tl)
  ```
  ```

Functions on binary trees

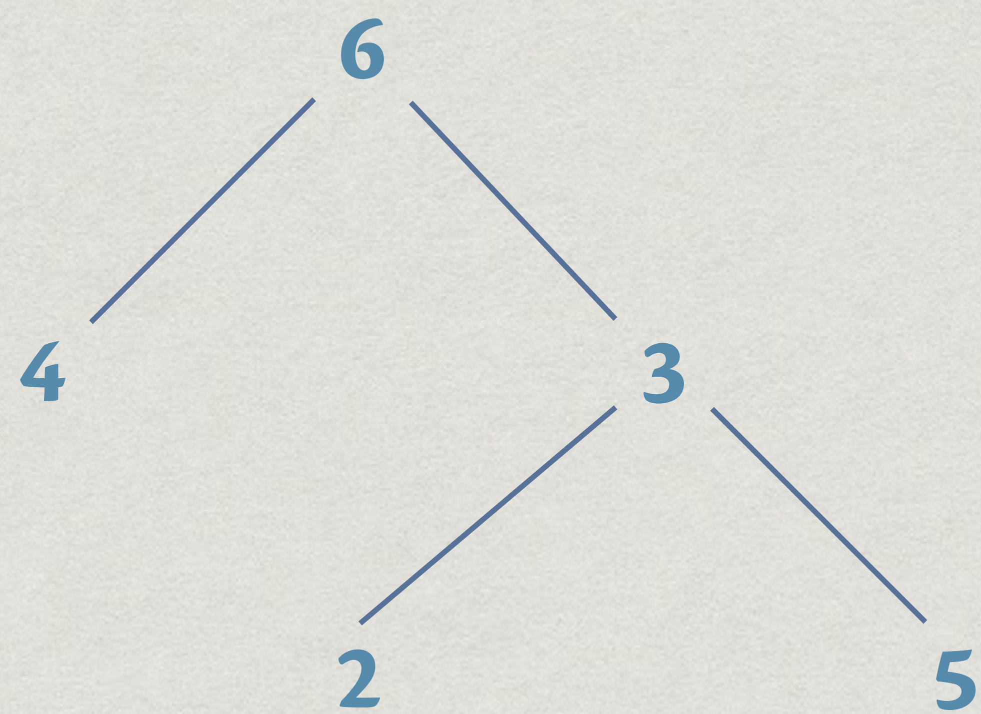
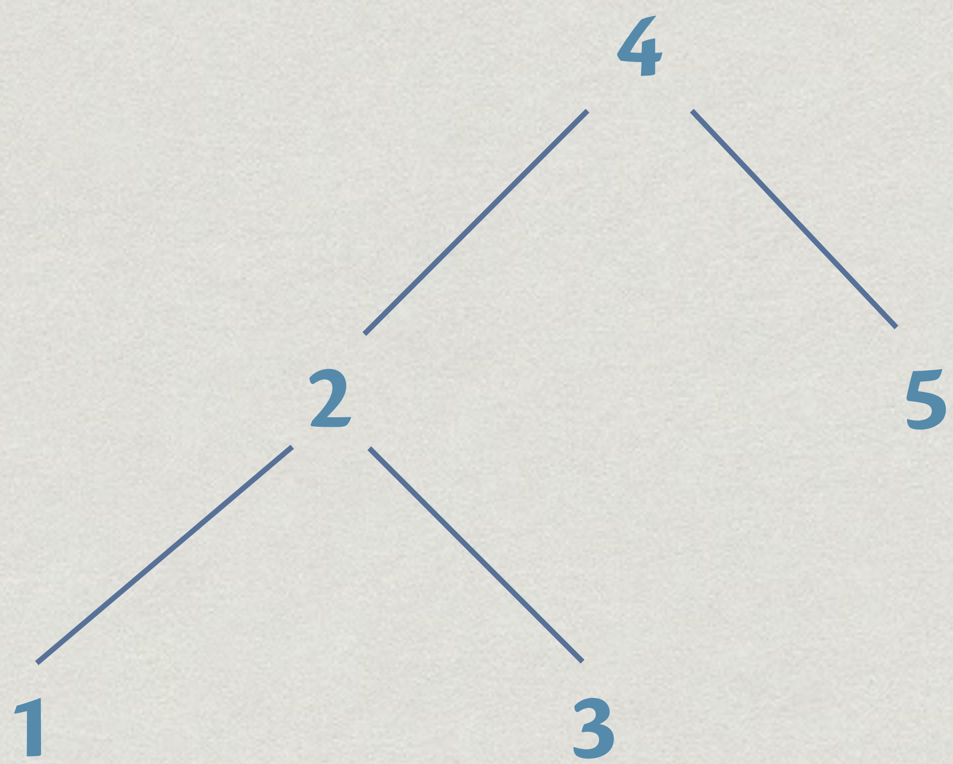
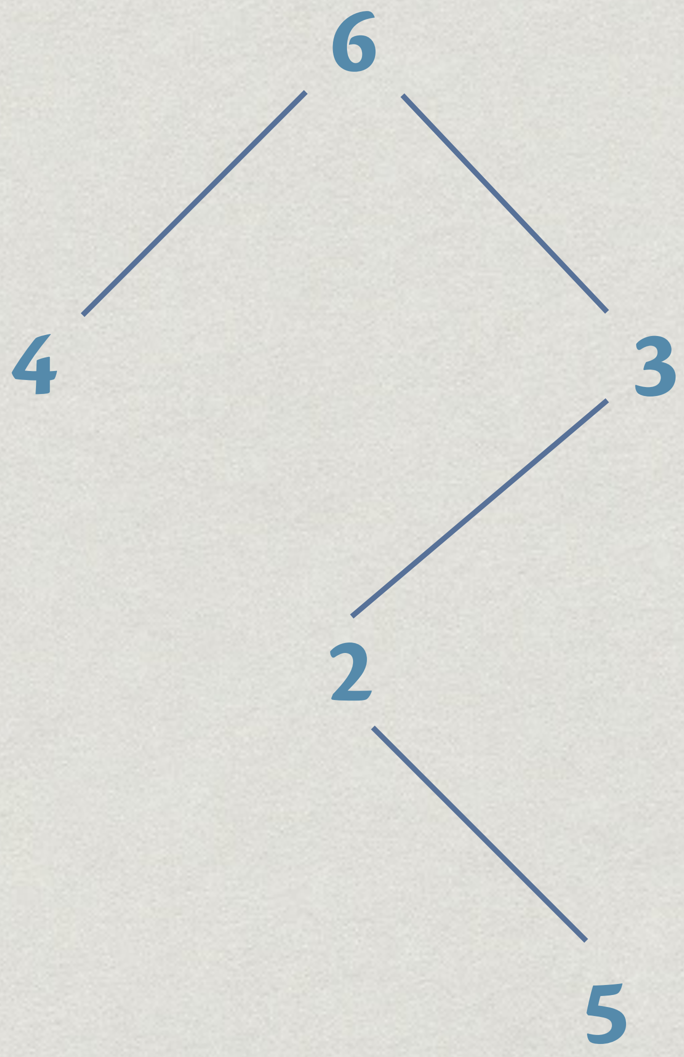
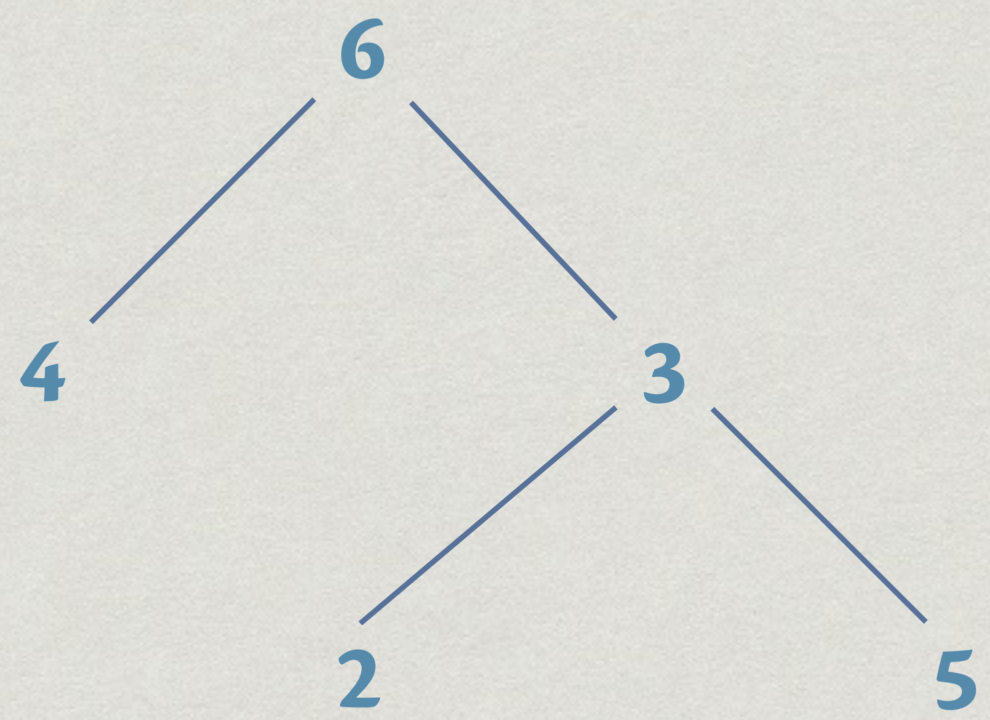
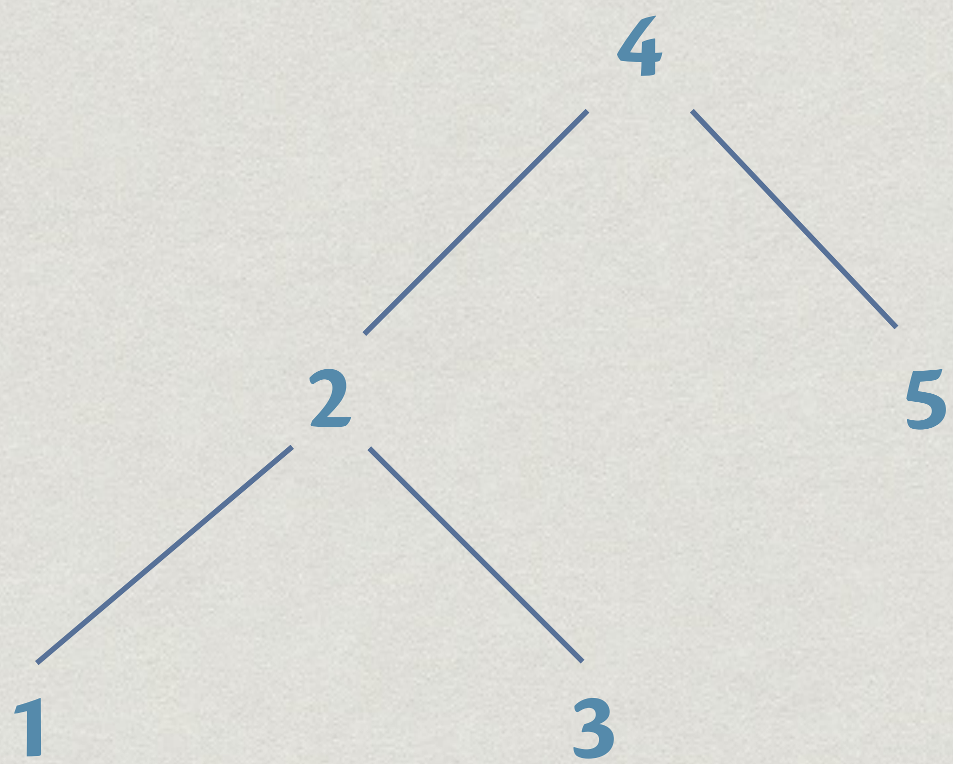
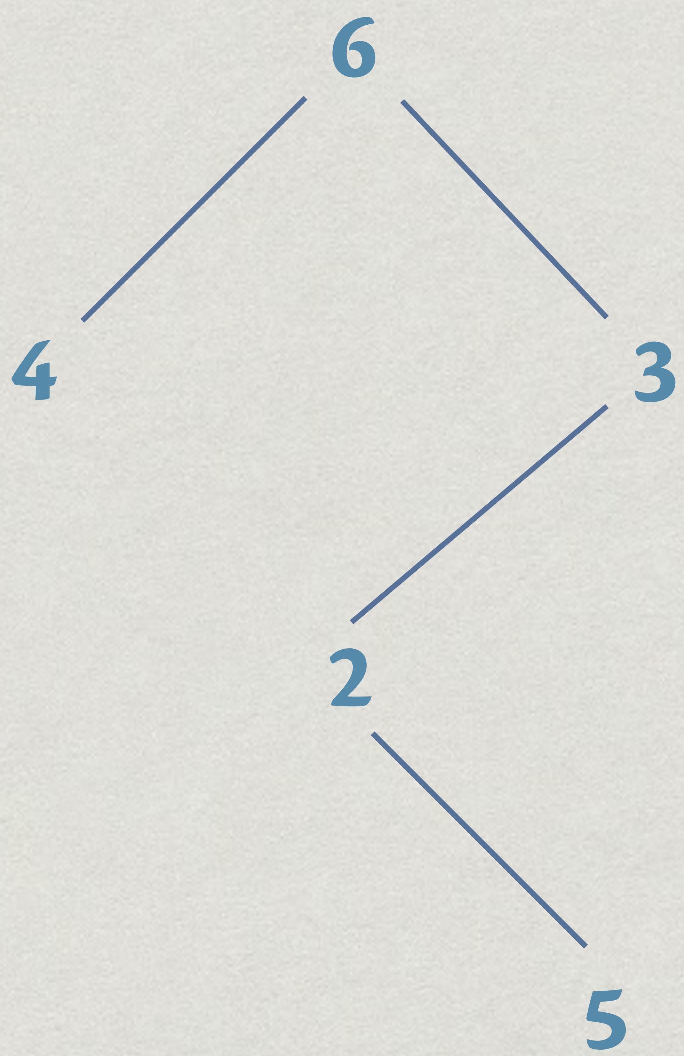


- `levels` - List nodes level by level, and from left to right within each level
- `levels` of the above tree - `[4,2,5,1,3]`

Functions on binary trees

- `levels t = concat (myLevels t)`
- ```
  myLevels :: BTree a -> [[a]]
  myLevels Nil                = []
  myLevels (Node t1 x t2) = [x]:
               join (myLevels t1)
                    (myLevels t2)
  ```
  ```

Functions on binary trees

* ```
  join :: [[a]] -> [[a]] -> [[a]]
  join [] yss               = yss
  join xss []               = xss
  join (xs:xss) (ys:yss)    = (xs ++ ys):
                                join xss yss
  ```
  ```

Showing trees

- ```
  data BTree a = Nil
               | Node (BTree a) a (BTree a)
       deriving (Eq, Show)
  ```
- Default `show` of trees is very hard to parse
- `show (Node (Node Nil 2 Nil) 3 (Node Nil 5 Nil)) = "Node (Node Nil 2 Nil) 3 (Node Nil 5 Nil)"`
  ```

A prettier show

- We want a better layout
- `tree1 = Node (Node Nil 4 Nil) 6 (Node (Node Nil 2 Nil) 3 (Node Nil 5 Nil))`
- Typing `tree1` in ghci should give us (each node on a line, and `2n` spaces before each node at level `n`)

```
6
  4
  3
    2
    5
```

A prettier show




```
6
   4
   3
      2
      5
```

A prettier show




```
4
  2
    1
    3
  5
```

A prettier show




```
6
  4
  3
    2
      *
      5
    *
```

A prettier show

- ```
  instance (Show a)    => Show (BTree a) where
      show t             = drawTree t ""
  ```

- ```
  drawTree :: (Show a) => BTree a ->
                          String -> String
  drawTree Nil spaces  = spaces ++ "*\n"
  ```

A prettier show

- ```
  instance (Show a) => Show (BTree a) where
      show t          = drawTree t ""
  ```

- ```
  drawTree (Node Nil x Nil) spaces
                   = spaces ++ show x ++ "\n"
  drawTree (Node tl x tr) spaces
                   = spaces++ show x ++ "\n"
                  ++ drawTree tl (' ':' ':spaces)
                  ++ drawTree tr (' ':' ':spaces)
  ```

Yet another show




```
6
|
+--4
|
`--3
   |
   +--2
   |
   `--5
```

Yet another show




```
4
|
+--2
|  |
|  +--1
|  |
|  `--3
|
`--5
```

Yet another show




```
6
|
+--4
|
`--3
   |
   +--2
   |  |
   |  +--*
   |  |
   |  `--5
   |
   `--*
```

Yet another show

- ```
  data Dir = LeftDir | RightDir
  type Path = [Dir]
  ```
- ```
  instance (Show a)    => Show (BTree a) where
      show t           = drawTree2 t []
  ```
- ```
  drawTree2 :: Show a => BTree a -> Path -> String
  ```
  ```

Yet another show

```
drawTree2 Nil path                          = numberLine path ++
                                                          "*\n"
drawTree2 (Node Nil x Nil) path             = numberLine path ++
                                                 show x ++ "\n"
drawTree2 (Node tl x tr) path               =
  numberLine path ++ show x ++ "\n" ++
  emptyLine pathl ++ "\n" ++ drawTree2 tl pathl ++
  emptyLine pathr ++ "\n" ++ drawTree2 tr pathr

   where
      pathl                                 = path ++ [LeftDir]
      pathr                                 = path ++ [RightDir]
```

Yet another show

- ```
  emptyLine :: Path -> String
  emptyLine []              = ""
  emptyLine [LeftDir]       = "|   "
  emptyLine [RightDir]      = "|   "
  emptyLine (LeftDir:ds)  = "|   " ++ emptyLine ds
  emptyLine (RightDir:ds)= "    " ++ emptyLine ds
  ```

- ```
  numberLine :: Path -> String
  numberLine []              = ""
  numberLine [LeftDir]       = "+   "
  numberLine [RightDir]      = "`   "
  numberLine (LeftDir:ds) = "|   " ++ numberLine ds
  numberLine (RightDir:ds)= "    " ++ numberLine ds
  ```

Summary

- Recursive datatypes are an important concept in Haskell
- A recursive datatype **T** is one which has some components of the same type **T**
- Two canonical and important examples of recursive datatypes – Lists and trees