Lenguajes, Tecnologías y Paradigmas de la programación (LTP)

Práctica 5: Listas y Tipos algebraicos (Parte 2: Tipos algebraicos)

UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Sergio Pérez
serperu@dsic.upv.es

SESIÓN 2: Tipos Algebraicos

OBJETIVOS DE LA PRÁCTICA

- Declarar tipos básicos mediante enumeración o renombramiento

- Definir tipos de datos recursivos y trabajar con ellos

Listas y tuplas

```
foo :: Int -> Int
    …
```

Listas y tuplas

```
foo :: Int -> Int
    …

foo ::  Int  -> Int
    …
```

Listas y tuplas

```
foo :: Int -> Int
    …

foo :: [Int] -> Int
    …
```

Listas y tuplas

```
foo :: Int -> Int
    …

foo :: [Int] -> Int
    …

foo ::   Int       -> Int
    …
```

Listas y tuplas

```
foo :: Int -> Int
    …

foo :: [Int] -> Int
    …

foo ::  (Int,Int)  -> Int
    …
```

Listas y tuplas

```
foo :: Int -> Int
    …

foo :: [Int] -> Int
    …

foo :: [(Int,Int)] -> Int
    …
```

Tipos Algebraicos: Renombramiento

Tipos que se definen en base a otros tipos existentes (sinónimos):

Tipos Algebraicos: Renombramiento

Tipos que se definen en base a otros tipos existentes (sinónimos):

```
type Person = String
type Book = String
```

Tipos Algebraicos: Renombramiento

Tipos que se definen en base a otros tipos existentes (sinónimos):

```
type Person = String
type Book = String
type Database = [(Person, Book)]
```

Tipos Algebraicos: Renombramiento

Tipos que se definen en base a otros tipos existentes (sinónimos):

```
type Person = String
type Book = String
type Database = [(Person, Book)]
```

NOTA: `String` también es un tipo de dato renombrado

```
type String = [Char]
```

Tipos Algebraicos: Enumeración

*Representa un tipo llamado **Color** que pueda ser de 4 tipos diferentes (**Blue**, **Green**, **Red**, **Yellow**)*

Tipos Algebraicos: Enumeración

*Representa un tipo llamado **Color** que pueda ser de 4 tipos diferentes (**Blue**, **Green**, **Red**, **Yellow**)*

```
data Color = Blue | Green | Red | Yellow
```

Tipos Algebraicos: Enumeración

*Representa un tipo llamado **Color** que pueda ser de 4 tipos diferentes (**Blue**, **Green**, **Red**, **Yellow**)*

```
data Color = Blue | Green | Red | Yellow

foo :: Int -> Color
foo x
```

Tipos Algebraicos: Enumeración

*Representa un tipo llamado **Color** que pueda ser de 4 tipos diferentes (**Blue**, **Green**, **Red**, **Yellow**)*

```haskell
data Color = Blue | Green | Red | Yellow

foo :: Int -> Color
foo x
  | mod x 4 == 0 = Blue
```

Tipos Algebraicos: Enumeración

*Representa un tipo llamado **Color** que pueda ser de 4 tipos diferentes (**Blue**, **Green**, **Red**, **Yellow**)*

```
data Color = Blue | Green | Red | Yellow

foo :: Int -> Color
foo x
  | mod x 4 == 0 = Blue
  | mod x 4 == 1 = Green …
```

Tipos Algebraicos: Tipos de datos recursivos

Veremos los tipos recursivos que se utilizan para representar árboles:

Tipos Algebraicos: Tipos de datos recursivos

Veremos los tipos recursivos que se utilizan para representar árboles:

- Árboles que almacenan valores en las hojas (`Tree a`)

```
data Tree a = Leaf a
```

Tipos Algebraicos: Tipos de datos recursivos

Veremos los tipos recursivos que se utilizan para representar árboles:

- Árboles que almacenan valores en las hojas (`Tree a`)

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

Tipos Algebraicos: Tipos de datos recursivos

Veremos los tipos recursivos que se utilizan para representar árboles:

- Árboles que almacenan valores en las hojas (`Tree a`)

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

Tipos Algebraicos: Tipos de datos recursivos

Veremos los tipos recursivos que se utilizan para representar árboles:

- Árboles que almacenan valores en las hojas (`Tree a`)

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

Tipo que usaremos a la parte derecha

Tipos Algebraicos: Tipos de datos recursivos

Veremos los tipos recursivos que se utilizan para representar árboles:

- Árboles que almacenan valores en las hojas (`Tree a`)

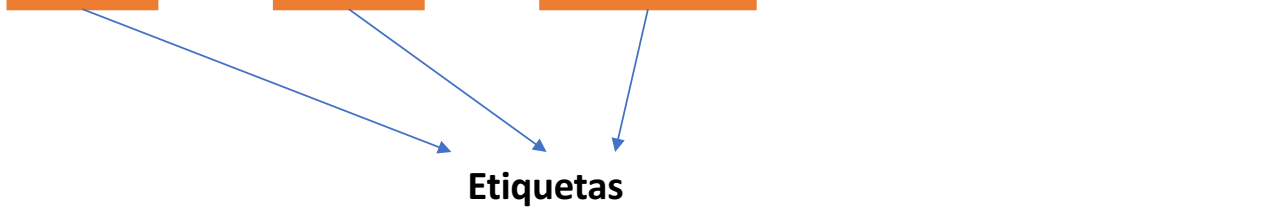
```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

Tipos Algebraicos: Tipos de datos recursivos

Veremos los tipos recursivos que se utilizan para representar árboles:

- Árboles que almacenan valores en las hojas (`Tree a`)

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



Tipos Algebraicos: Tipos de datos recursivos

Veremos los tipos recursivos que se utilizan para representar árboles:

- Árboles que almacenan valores en las hojas (`Tree a`)

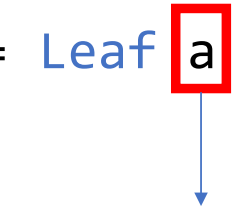
```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

Tipos Algebraicos: Tipos de datos recursivos

Veremos los tipos recursivos que se utilizan para representar árboles:

- Árboles que almacenan valores en las hojas (`Tree a`)

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



Valor del tipo indicado

Tipos Algebraicos: Tipos de datos recursivos

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

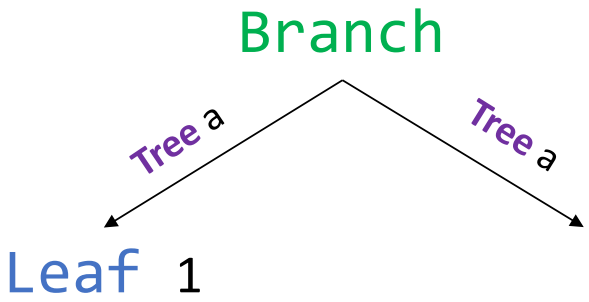
Tipos Algebraicos: Tipos de datos recursivos

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

```
Leaf 1
```

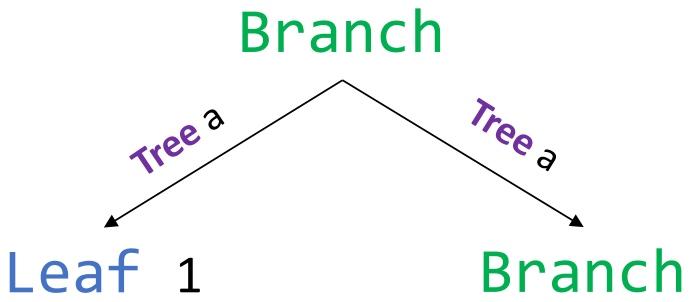
Tipos Algebraicos: Tipos de datos recursivos

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



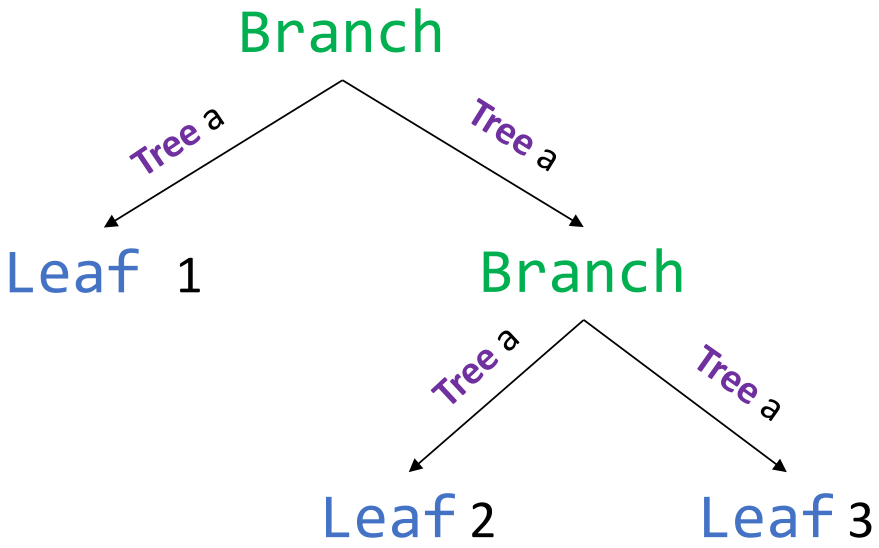
Tipos Algebraicos: Tipos de datos recursivos

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



Tipos Algebraicos: Tipos de datos recursivos

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



Tipos Algebraicos: Tipos de datos recursivos

Veremos los tipos recursivos que se utilizan para representar árboles:

- Árboles que almacenan valores en las hojas (`Tree a`)

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

- Árboles que almacenan valores en los nodos (`BinTreeInt`)

```
data BinTreeInt = Void
```

Tipos Algebraicos: Tipos de datos recursivos

Veremos los tipos recursivos que se utilizan para representar árboles:

- Árboles que almacenan valores en las hojas (`Tree a`)

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

- Árboles que almacenan valores en los nodos (`BinTreeInt`)

```
data BinTreeInt = Void | Node Int BinTreeInt BinTreeInt
```

Tipos Algebraicos: Tipos de datos recursivos

Veremos los tipos recursivos que se utilizan para representar árboles:

- Árboles que almacenan valores en las hojas (`Tree a`)

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

- Árboles que almacenan valores en los nodos (`BinTreeInt`)

```
data BinTreeInt = Void | Node Int BinTreeInt BinTreeInt
```

Tipos Algebraicos: Tipos de datos recursivos

Veremos los tipos recursivos que se utilizan para representar árboles:

- Árboles que almacenan valores en las hojas (`Tree a`)

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

- Árboles que almacenan valores en los nodos (`BinTreeInt`)

```
data BinTreeInt = Void | Node Int BinTreeInt BinTreeInt
```


Usamos un tipo definido, no necesitamos indicarlo en la parte izquierda

Tipos Algebraicos: Tipos de datos recursivos

¿Cómo funciona un árbol binario de enteros?

Tipos Algebraicos: Tipos de datos recursivos

¿Cómo funciona un árbol binario de enteros?

- Cada nodo tiene un entero

Tipos Algebraicos: Tipos de datos recursivos

¿Cómo funciona un árbol binario de enteros?

- Cada nodo tiene un entero
- Los nodos se insertan desde la raíz
- Cuando inserto un nuevo entero:

Tipos Algebraicos: Tipos de datos recursivos

¿Cómo funciona un árbol binario de enteros?

- Cada nodo tiene un entero
- Los nodos se insertan desde la raíz
- Cuando inserto un nuevo entero:
  - Si el nuevo entero es menor que el nodo que estoy mirando, lo inserto en la rama izquierda

Tipos Algebraicos: Tipos de datos recursivos

¿Cómo funciona un árbol binario de enteros?

- Cada nodo tiene un entero
- Los nodos se insertan desde la raíz
- Cuando inserto un nuevo entero:
  - Si el nuevo entero es menor que el nodo que estoy mirando, lo inserto en la rama izquierda
  - Si el nuevo entero es mayor que el nodo que estoy mirando, lo inserto en la rama derecha

Tipos Algebraicos: Tipos de datos recursivos

```
data BinTreeInt = Void | Node Int BinTreeInt BinTreeInt
```

Tipos Algebraicos: Tipos de datos recursivos

```
data BinTreeInt = Void | Node Int BinTreeInt BinTreeInt
```

```
Void
```

Tipos Algebraicos: Tipos de datos recursivos

```
data BinTreeInt = Void | Node Int BinTreeInt BinTreeInt
```

Void

[5,3,8,6]

Tipos Algebraicos: Tipos de datos recursivos

```
data BinTreeInt = Void | Node Int BinTreeInt BinTreeInt
```

Void

[5,3,8,6]

Tipos Algebraicos: Tipos de datos recursivos

```
data BinTreeInt = Void | Node Int BinTreeInt BinTreeInt
```

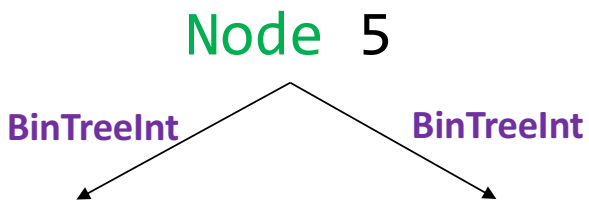
```
Node 5
```

[5,3,8,6]

Tipos Algebraicos: Tipos de datos recursivos

```
data BinTreeInt = Void | Node Int BinTreeInt BinTreeInt
```

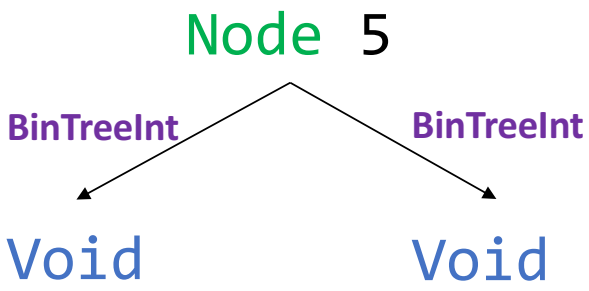
[5,3,8,6]



Tipos Algebraicos: Tipos de datos recursivos

```
data BinTreeInt = Void | Node Int BinTreeInt BinTreeInt
```

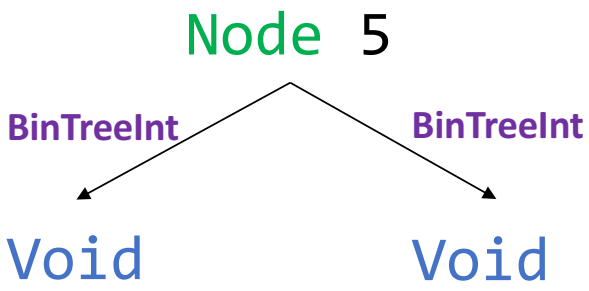
[5,3,8,6]



Tipos Algebraicos: Tipos de datos recursivos

```
data BinTreeInt = Void | Node Int BinTreeInt BinTreeInt
```

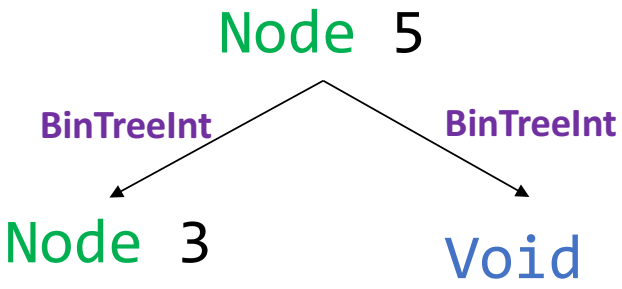
[5,3,8,6]



Tipos Algebraicos: Tipos de datos recursivos

```
data BinTreeInt = Void | Node Int BinTreeInt BinTreeInt
```

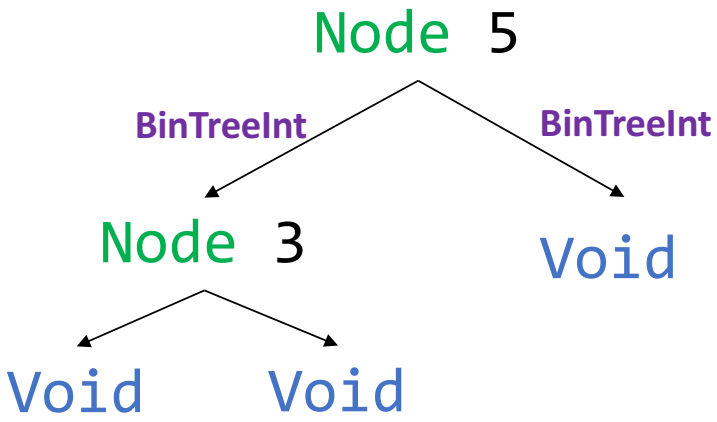
[5,3,8,6]



Tipos Algebraicos: Tipos de datos recursivos

```
data BinTreeInt = Void | Node Int BinTreeInt BinTreeInt
```

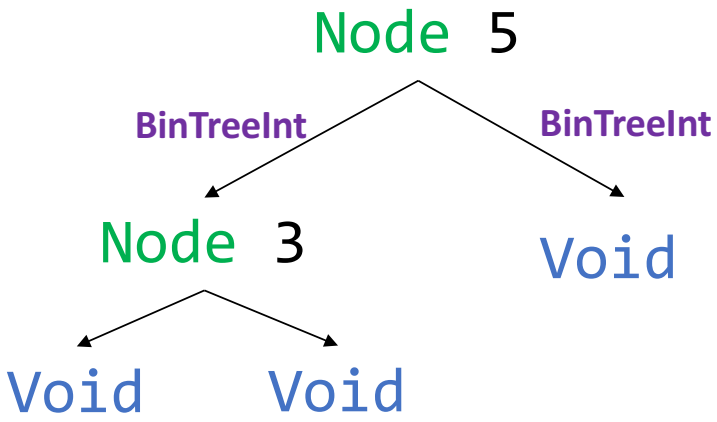
[5,3,8,6]



Tipos Algebraicos: Tipos de datos recursivos

```
data BinTreeInt = Void | Node Int BinTreeInt BinTreeInt
```

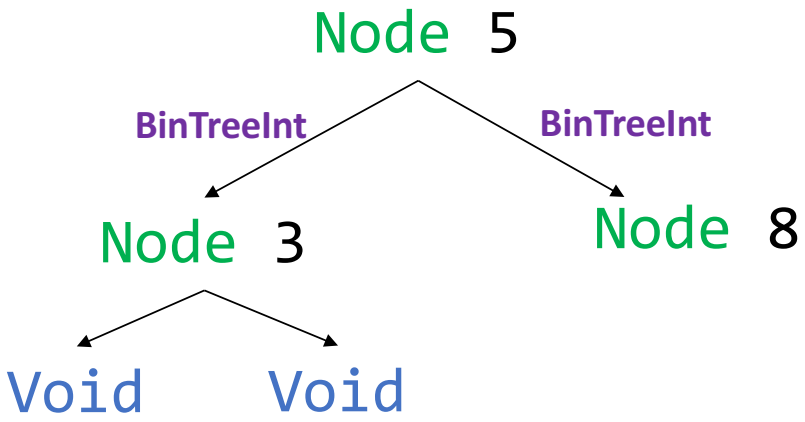
[5,3,8,6]



Tipos Algebraicos: Tipos de datos recursivos

```
data BinTreeInt = Void | Node Int BinTreeInt BinTreeInt
```

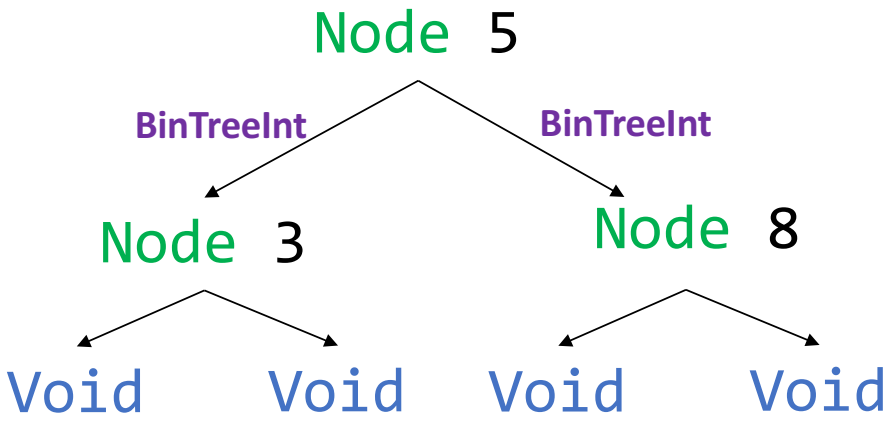
[5,3,8,6]



Tipos Algebraicos: Tipos de datos recursivos

```
data BinTreeInt = Void | Node Int BinTreeInt BinTreeInt
```

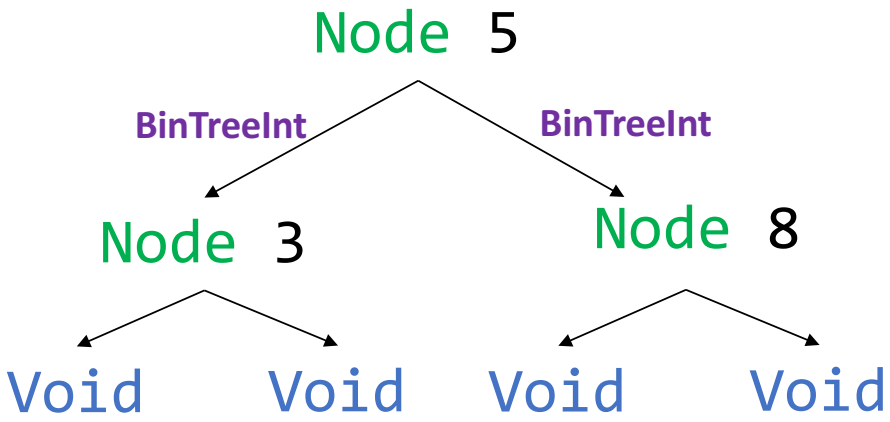
[5,3,8,6]



Tipos Algebraicos: Tipos de datos recursivos

```
data BinTreeInt = Void | Node Int BinTreeInt BinTreeInt
```

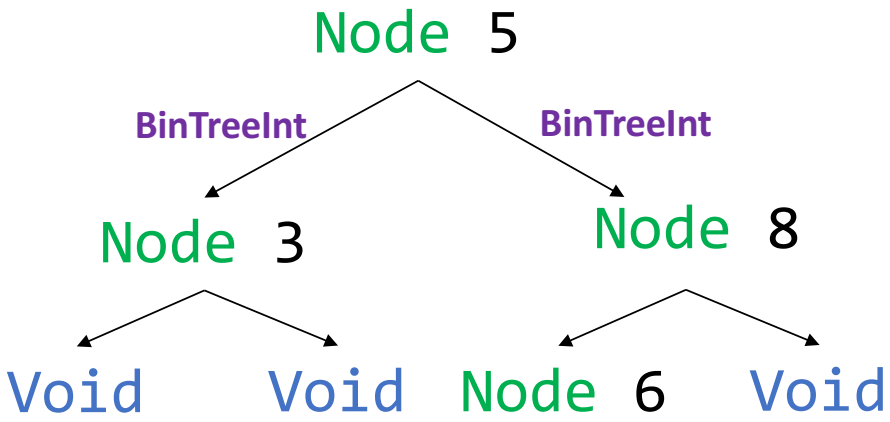
[5,3,8,6]



Tipos Algebraicos: Tipos de datos recursivos

```
data BinTreeInt = Void | Node Int BinTreeInt BinTreeInt
```

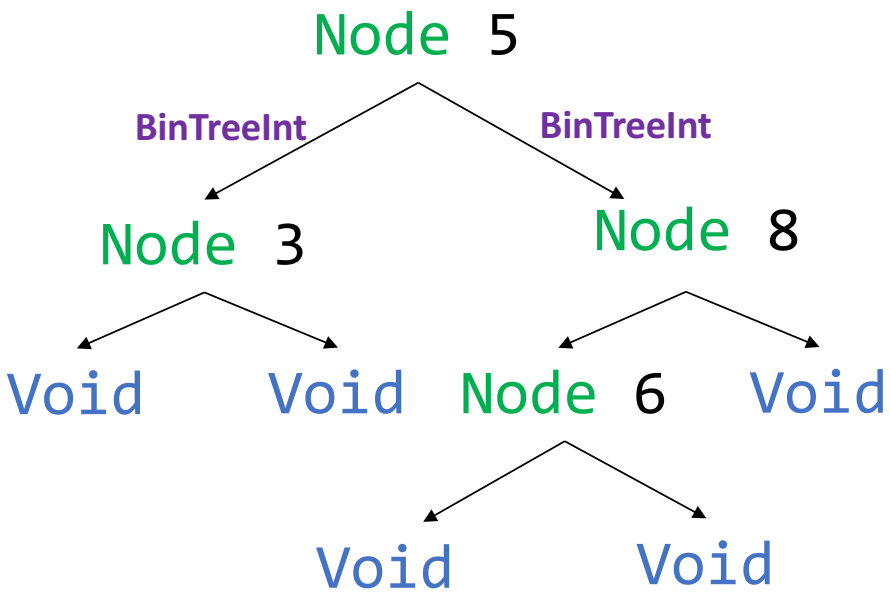
[5,3,8,6]



Tipos Algebraicos: Tipos de datos recursivos

```
data BinTreeInt = Void | Node Int BinTreeInt BinTreeInt
```

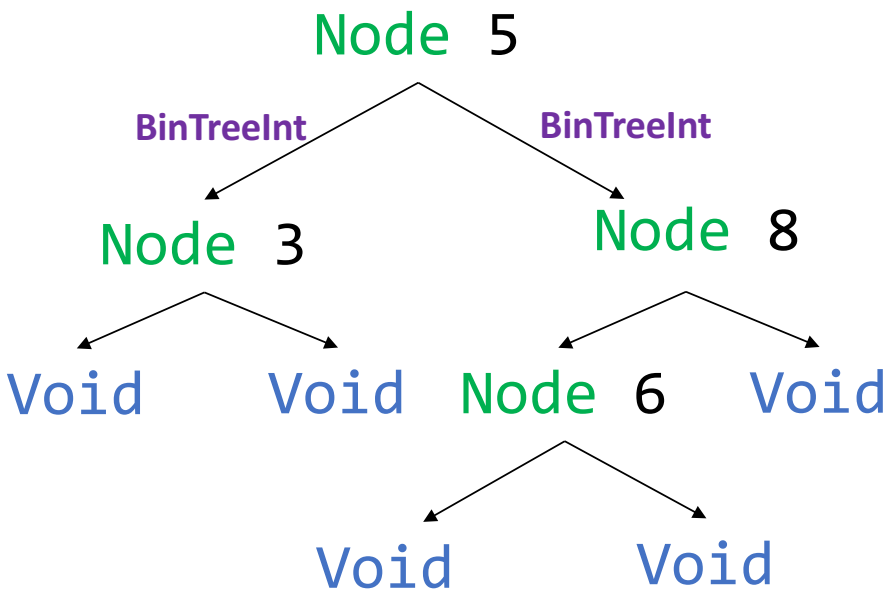
[5,3,8,6]



Tipos Algebraicos: Tipos de datos recursivos

```
data BinTreeInt = Void | Node Int BinTreeInt BinTreeInt
```

[5,3,8,6]



Ejercicios

- Ejercicios Parte 2:
  - Ejercicios 11 - 16

- Ampliación Parte 2:
  - Ejercicios 20, 21 y 22

Ejercicio 12

```haskell
module Symmetric where
   data Tree a = Leaf a | Branch (Tree a) (Tree a) deriving Show
```

Ejercicio 12

```haskell
module Symmetric where
  data Tree a = Leaf a | Branch (Tree a) (Tree a) deriving Show
  symmetric :: Tree a -> Tree a
```

Ejercicio 12

```haskell
module Symmetric where
   data Tree a = Leaf a | Branch (Tree a) (Tree a) deriving Show
   symmetric :: Tree a -> Tree a
   symmetric (Leaf l)
```

Ejercicio 12

```haskell
module Symmetric where
   data Tree a = Leaf a | Branch (Tree a) (Tree a) deriving Show
   symmetric :: Tree a -> Tree a
   symmetric (Leaf l) = (Leaf l)
```

Ejercicio 12

```haskell
module Symmetric where
   data Tree a = Leaf a | Branch (Tree a) (Tree a) deriving Show
   symmetric :: Tree a -> Tree a
   symmetric (Leaf l) = (Leaf l)
   symmetric (Branch a b)
```

Ejercicio 12

```haskell
module Symmetric where
  data Tree a = Leaf a | Branch (Tree a) (Tree a) deriving Show
  symmetric :: Tree a -> Tree a
  symmetric (Leaf l) = (Leaf l)
  symmetric (Branch a b) = Branch b a
```

Ejercicio 12

```haskell
module Symmetric where
  data Tree a = Leaf a | Branch (Tree a) (Tree a) deriving Show
  symmetric :: Tree a -> Tree a
  symmetric (Leaf l) = (Leaf l)
  symmetric (Branch a b) = Branch (symmetric b) (symmetric a)
```

Ejercicio 12

```haskell
module Symmetric where
   data Tree a = Leaf a | Branch (Tree a) (Tree a) deriving Show
   symmetric :: Tree a -> Tree a
   symmetric (Leaf l) = (Leaf l)
   symmetric (Branch a b) = Branch (symmetric b) (symmetric a)
```

```
Prelude> :l Symmetric
[1 of 1] Compiling Symmetric       (Symmetric, interpreted)
Ok, one module loaded.
```

Ejercicio 12

```haskell
module Symmetric where
   data Tree a = Leaf a | Branch (Tree a) (Tree a) deriving Show
   symmetric :: Tree a -> Tree a
   symmetric (Leaf l) = (Leaf l)
   symmetric (Branch a b) = Branch (symmetric b) (symmetric a)
```

```
Prelude> :l Symmetric
[1 of 1] Compiling Symmetric       (Symmetric, interpreted)
Ok, one module loaded.

*Symmetric> symmetric (Branch (Branch (Leaf 1) (Leaf 3)) (Leaf 2))
```

Ejercicio 12

```
module Symmetric where
   data Tree a = Leaf a | Branch (Tree a) (Tree a) deriving Show
   symmetric :: Tree a -> Tree a
   symmetric (Leaf l) = (Leaf l)
   symmetric (Branch a b) = Branch (symmetric b) (symmetric a)
```

```
Prelude> :l Symmetric
[1 of 1] Compiling Symmetric       (Symmetric, interpreted)
Ok, one module loaded.

*Symmetric> symmetric (Branch (Branch (Leaf 1) (Leaf 3)) (Leaf 2))
```

```
Branch (Leaf 2) (Branch (Leaf 3) (Leaf 1))
```