Estructuras de datos

Clases teóricas

por Pablo E. "Fidel" Martínez López

7. Tipos Abstractos de Datos III

Heaps, BSTs y AVLs

Repaso

Tipos abstractos de datos

- Los tipos abstractos de datos (TADs)
  - Quedan definidos por su ***interfaz***
  - Induce roles en la manera de usarlo
    - Diseñador, usuario, implementador
    - Cada rol tiene diferentes obligaciones y responsabilidades
  - Requieren ciertas herramientas para implementarlos
    - Invariantes de representación, eficiencia

Eficiencia

- Para medir eficiencia se usan modelos especiales
  - Modelo de ***peor caso*** para medir operaciones
    - En base al comportamiento del peor caso
    - En función de la cantidad de elementos de la estructura
    - Con una medición gruesa (sin detalles)
  - Clasificación
    - Constante, siempre el mismo costo, O(1)
    - Lineal, solo operaciones constantes por elemento, O(n)
    - Cuadrática, hasta operaciones lineales por elemento, O(n^2)

Tipos abstractos de datos

- Existen TADs clásicos que hay que conocer
  - Stacks, Queues, Sets, PriorityQueues, Maps, Multisets
- Al estudiarlos, aprendemos las herramientas necesarias
  - Como usuario
    - Usar la intefaz sin conocer implementaciones
  - Como implementador
    - Elección de representaciones eficaces
    - Formas de invariantes útiles para mejorar eficiencia
    - Mediciones de eficiencia para tener alternativas

Tipos abstractos de datos

- Stacks
  - el último que entra es el primero que sale
- Queues
  - el primero que entra es el primero que sale
- Sets
  - indica si un elemento fue agregado o no

Tipos abstractos de datos

- PriorityQueues
  - el próximo que sale es el mínimo (de máxima prioridad)
- Maps (o Diccionario)
  - sale la clave asociada a un valor, si existe
- Multisets
  - indica la cantidad de veces que se agregó un elemento

Mejoras de eficiencia

Implementaciones lineales

- Vimos implementaciones lineales para todos los TADs
  - Usan listas
  - Mayoría de operaciones de O(n)
  - ¿Se puede mejorar? No a O(1)
  - Debería haber algo intermedio entre O(n) y O(1)…
  - ¿Qué costo tienen las búsquedas en árboles?
    - Ej.: en Dungeons…

Eficiencia: clasificación

- Constante, O(1)
  - siempre el mismo costo
- Lineal, O(n)
  - operaciones constantes por cada elemento
- Cuadrática, O(n^2)
  - hasta operaciones lineales por cada elemento

Eficiencia: clasificación

- Constante, O(1)
  - siempre el mismo costo
- ??

¿Habrá algo en medio?

- Lineal, O(n)
  - operaciones constantes por cada elemento
- ??

¿Habrá algo en medio?

- Cuadrática, O(n^2)
  - hasta operaciones lineales por cada elemento

Búsqueda en árboles

```
data Dir = Izq | Der
data Objeto = Armadura | Escudo | Maza | …
data Dungeon = Armario
             | Habitacion Objeto Dungeon
                                 Dungeon
```

- ¿Qué costo tiene una búsqueda en un árbol?

```
hayOroEn :: Dungeon -> [Dir] -> Bool
hayOroEn Armario                  _      = False
hayOroEn (Habitacion obj _  _ )  []     = esOro obj
hayOroEn (Habitacion _   d1 d2) (d:ds) = case d of
                                              Izq -> hayOroEn ds d1
                                              Der -> hayOroEn ds d2
```

  - ¿Cuál es el peor caso?

Búsqueda en árboles

```
data Dir = Izq | Der
data Objeto = Armadura | Escudo | Maza | …
data Dungeon = Armario
             | Habitacion Objeto Dungeon
                                 Dungeon
```

- ¿Qué costo tiene una búsqueda en un árbol?

```
hayOroEn :: Dungeon -> [Dir] -> Bool
hayOroEn Armario                      _      = False
hayOroEn (Habitacion obj _  _ ) []     = esOro obj
hayOroEn (Habitacion _   d1 d2) (d:ds) = case d of
                                            Izq -> hayOroEn ds d1
                                            Der -> hayOroEn ds d2
```

  - ¿Cuál es el peor caso?
    - Se recorre UNA rama del árbol completa (PARTE de la lista)

Búsqueda en árboles

```
data Dir = Izq | Der
data Objeto = Armadura | Escudo | Maza | …
data Dungeon = Armario
             | Habitacion Objeto Dungeon
                                 Dungeon
```

- ¿Qué costo tiene una búsqueda en un árbol?

```
hayOroEn :: Dungeon -> [Dir] -> Bool
hayOroEn Armario                    _      = False
hayOroEn (Habitacion obj _  _ )     []     = esOro obj
hayOroEn (Habitacion _   d1 d2)   (d:ds)   = case d of
                                               Izq -> hayOroEn ds d1
                                               Der -> hayOroEn ds d2
```

  - ¿Cuál es el peor caso?
    - Se recorre UNA rama del árbol completa (PARTE de la lista)
    - ¿Y el resto del árbol?

Búsqueda en árboles

```
data Dir = Izq | Der
data Objeto = Armadura | Escudo | Maza | …
data Dungeon = Armario
             | Habitacion Objeto Dungeon
                                 Dungeon
```

- ¿Qué costo tiene una búsqueda en un árbol?

```
hayOroEn :: Dungeon -> [Dir] -> Bool
hayOroEn Armario                  _      = False
hayOroEn (Habitacion obj _  _ )   []     = esOro obj
hayOroEn (Habitacion _   d1 d2)  (d:ds)  = case d of
                                              Izq -> hayOroEn ds d1
                                              Der -> hayOroEn ds d2
```

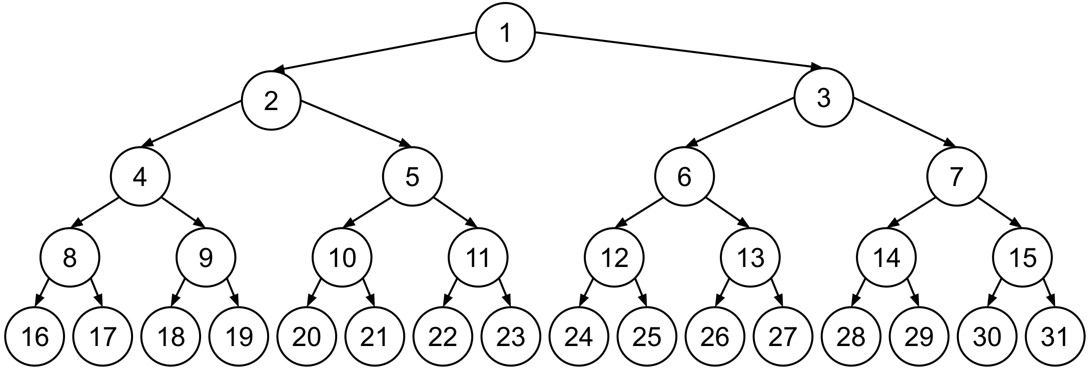
  - ¿Cuál es el peor caso?
    - Se recorre UNA rama del árbol completa (PARTE de la lista)
    - ¿Y el resto del árbol? No se recorre
    - ¿Quién es el n en este caso?

Una nueva categoría para costos

- Suponiendo que n es la cantidad de elementos del árbol
  - ¿Qué cantidad de elementos tiene una rama?

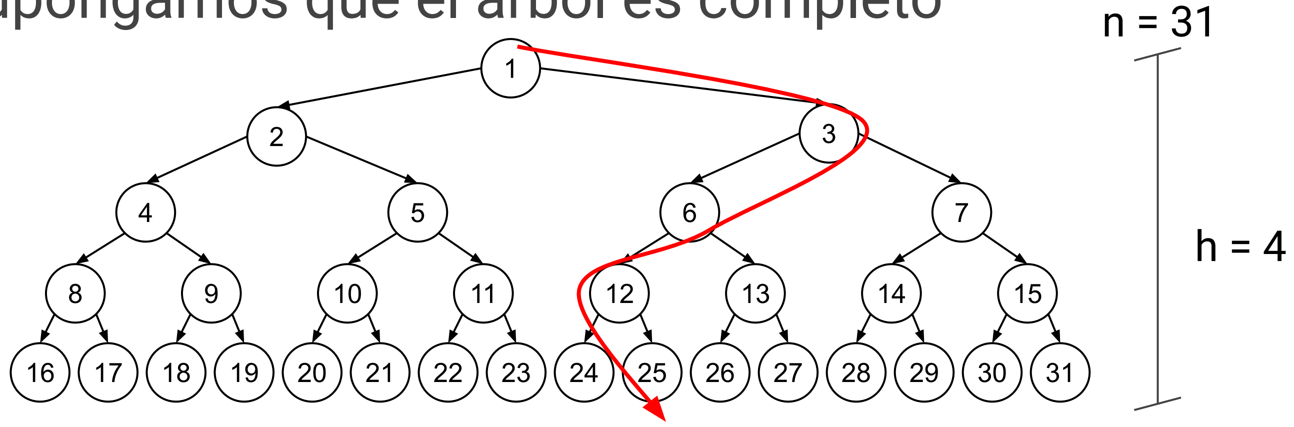
Una nueva categoría para costos

- Suponiendo que n es la cantidad de elementos del árbol
  - ¿Qué cantidad de elementos tiene una rama? Digamos, h
  - Supongamos que el árbol es completo



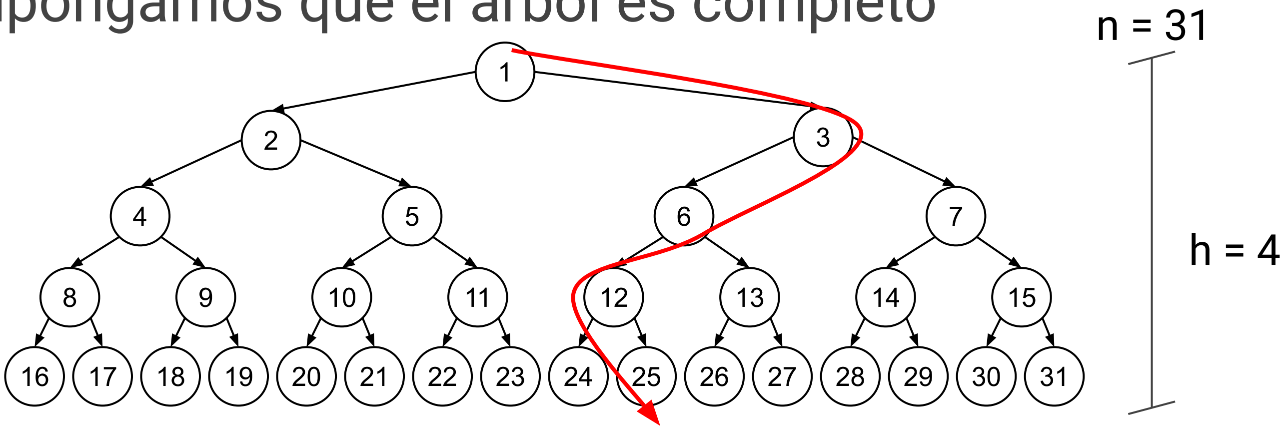
Una nueva categoría para costos

- Suponiendo que n es la cantidad de elementos del árbol
  - ¿Qué cantidad de elementos tiene una rama? Digamos, h
  - Supongamos que el árbol es completo



Una nueva categoría para costos

- Suponiendo que n es la cantidad de elementos del árbol
  - ¿Qué cantidad de elementos tiene una rama? Digamos, h
  - Supongamos que el árbol es completo



$$n = 2^h - 1$$

Una nueva categoría para costos

- Suponiendo que n es la cantidad de elementos del árbol
  - ¿Qué cantidad de elementos tiene una rama? Digamos, h
  - Supongamos que el árbol es completo: n = 2^h - 1
    - Entonces, ¿cuánto vale h?
    - El número h tal que 2^h es igual a n

Una nueva categoría para costos

- Suponiendo que n es la cantidad de elementos del árbol
  - ¿Qué cantidad de elementos tiene una rama? Digamos, h
  - Supongamos que el árbol es completo: $n = 2^h - 1$
    - Entonces, ¿cuánto vale h?
    - El número h tal que 2^h es igual a n
      - Se llama “logaritmo en base 2”
      - Notaremos log para decir “logaritmo en base 2”

Una nueva categoría para costos

- Suponiendo que n es la cantidad de elementos del árbol
  - ¿Qué cantidad de elementos tiene una rama? Digamos, h
  - Supongamos que el árbol es completo: n = 2^h - 1
    - Entonces, ¿cuánto vale h?
    - El número h tal que 2^h es igual a n
      - Se llama “logaritmo en base 2”
      - Notaremos log para decir “logaritmo en base 2”
    - h = log n
      - h = cuántas veces hay que multiplicar por 2 para dar n

Una nueva categoría para costos

- Suponiendo que n es la cantidad de elementos del árbol
  - ¿Qué cantidad de elementos tiene una rama? Digamos, h
  - Supongamos que el árbol es completo: n = 2^h - 1
    - Entonces, ¿cuánto vale h?
    - h = log n
      - h = cuántas veces hay que multiplicar por 2 para dar n

```
cuantasVecesMultiplicarPor2Para :: Int -> Int
cuantasVecesMultiplicarPor2Para 1 = 0   -- PRECOND: n>0
cuantasVecesMultiplicarPor2Para n =
       1 + cuantasVecesMultiplicarPor2Para (div n 2)
```

Una nueva categoría para costos

- Suponiendo que n es la cantidad de elementos del árbol
  - ¿Qué cantidad de elementos tiene una rama? Digamos, h
  - Supongamos que el árbol es completo: $n = 2^h - 1$
    - Entonces, ¿cuánto vale h?
    - $h = \log n$
      - h = cuántas veces hay que multiplicar por 2 para dar n

```
logBase2 :: Int -> Int  -- PRECOND: n>0
logBase2 1 = 0
logBase2 n = 1 + logBase2 (div n 2)
```

Logaritmos

- Logaritmo base 2, h = log n
  - h = cuántas veces hay que multiplicar por 2 para dar n
  - cuántos dígitos se precisan para escribir n en binario
  - en qué nivel del árbol está el elemento numerado n

```
logBase2 :: Int -> Int  -- PRECOND: n>0
logBase2 1 = 0
logBase2 n = 1 + logBase2(div n b)
```

Logaritmos

- Logaritmo base 2, h = log n
  - h = cuántas veces hay que multiplicar por 2 para dar n
  - cuántos dígitos se precisan para escribir n en binario
  - en qué nivel del árbol está el elemento numerado n
- Logaritmo base b, h = $\log_b$ n
  - h = cuántas veces hay que multiplicar por b para dar n

```
logBase :: Int -> Int -> Int  -- PRECOND: b,n>0
logBase b 1 = 0
logBase b n = 1 + logBase b (div n b)
```

Eficiencia: clasificación

- Constante, O(1)
  - siempre el mismo costo
- ??

¿Habrá algo en medio?

- Lineal, O(n)
  - operaciones constantes por cada elemento
- ??

¿Habrá algo en medio?

- Cuadrática, O(n^2)
  - hasta operaciones lineales por cada elemento

Eficiencia: clasificación

- Constante, O(1)
  - siempre el mismo costo
- Logarítmica, O(log n)
  - se recorre una rama de un árbol balanceado
- Lineal, O(n)
  - operaciones constantes por cada elemento
- ??
- Cuadrática, O(n^2)
  - hasta operaciones lineales por cada elemento

¡Hay algo en medio!

¿Habrá algo en medio?

Eficiencia: clasificación

- Constante, O(1)
  - siempre el mismo costo
- Logarítmica, O(log n)
  - se recorre una rama de un árbol balanceado
- Lineal, O(n)
  - operaciones constantes por cada elemento
- “Eneloguene”, O(n log n)
  - hasta operaciones logarítmicas por cada elemento
- Cuadrática, O(n^2)
  - hasta operaciones lineales por cada elemento

¡Hay algo en medio!

¡Hay algo en medio!

Búsqueda en árboles

```
data Dir = Izq | Der
data Objeto = Armadura | Escudo | Maza | …
data Dungeon = Armario
             | Habitacion Objeto Dungeon
                                 Dungeon
```

- ¿Qué costo tiene una búsqueda en un árbol?

```
hayOroEn :: Dungeon -> [Dir] -> Bool
hayOroEn Armario                  _      = False
hayOroEn (Habitacion obj _  _ )   []     = esOro obj
hayOroEn (Habitacion _   d1 d2)   (d:ds) = case d of
                                              Izq -> hayOroEn ds d1
                                              Der -> hayOroEn ds d2
```

  - Costo logarítmico, O(log n), si el árbol está balanceado
  - ¿Y si no lo está?

Búsqueda en árboles

```
data Dir = Izq | Der
data Objeto = Armadura | Escudo | Maza | …
data Dungeon = Armario
             | Habitacion Objeto Dungeon
                                 Dungeon
```

- ¿Qué costo tiene una búsqueda en un árbol?

```
hayOroEn :: Dungeon -> [Dir] -> Bool
hayOroEn Armario                   _      = False
hayOroEn (Habitacion obj _  _ )    []     = esOro obj
hayOroEn (Habitacion _   d1 d2)   (d:ds)  = case d of
                                             Izq -> hayOroEn ds d1
                                             Der -> hayOroEn ds d2
```

  - Costo logarítmico, O(log n), si el árbol está balanceado
  - ¿Y si no lo está?
    - ¡Sigue siendo lineal en peor caso!
    - Veremos de agregar condiciones para balanceo…

Búsqueda en árboles

```
data Dir = Izq | Der
data Objeto = Armadura | Escudo | Maza | …
data Dungeon = Armario
             | Habitacion Objeto Dungeon
                                 Dungeon
```

- ¿Qué costo tiene una búsqueda en un árbol?

```
hayOroEn :: Dungeon -> [Dir] -> Bool
hayOroEn Armario                   _      = False
hayOroEn (Habitacion obj _  _ )  []     = esOro obj
hayOroEn (Habitacion _   d1 d2) (d:ds) = case d of
                                            Izq -> hayOroEn ds d1
                                            Der -> hayOroEn ds d2
```

  - Costo logarítmico, O(log n), si el árbol está balanceado
  - ¿Y si no lo está?
    - ¡Sigue siendo lineal en peor caso!
    - Veremos de agregar condiciones para balanceo… ¿Cómo?

Búsqueda en árboles

```
data Dir = Izq | Der
data Objeto = Armadura | Escudo | Maza | …
data Dungeon = Armario
             | Habitacion Objeto Dungeon
                                 Dungeon
```

- ¿Qué costo tiene una búsqueda en un árbol?

```
hayOroEnAlgunoEn :: [Dir] -> [Dungeon] -> Bool
hayOroEnAlgunoEn ds []      = False
hayOroEnAlgunoEn ds (m:ms) =
                hayOroEn m ds || hayOroEnAlgunoEn ds ms
```

  - n, tamaño máximo de un árbol; t, cantidad de árboles
  - Cada operación es logarítmica en un árbol: O(log n)
  - Hay t operaciones…
  - El costo es O(t log n)
    - Más que lineal en t, menos que t^2

Nuevas implementaciones: BSTs

Buscando implementaciones logarítmicas

- ¿Podemos implementar **`Set`**s con costos logarítmicos?

```
module Set (Set, emptyS, addS, belongs
               , removeS, set2list) where

data Set a

emptyS    :: Set a
addS      :: Ord a => a -> Set a -> Set a
belongs   :: Ord a => a -> Set a -> Bool
removeS   :: Ord a => a -> Set a -> Set a
set2list  :: Ord a => Set a -> [a]
```

Buscando implementaciones logarítmicas

- ¿Podemos implementar `Set`s con costos logarítmicos?
- Precisamos que
  - Los elementos estén ordenados
  - Los elementos estén organizados en un árbol
  - ¿Cómo ordenar un árbol?
  - ¿Cómo garantizar las condiciones de orden?

Buscando implementaciones logarítmicas

- ¿Podemos implementar `Set`s con costos logarítmicos?
- Precisamos que
  - Los elementos estén ordenados
  - Los elementos estén organizados en un árbol
  - ¿Cómo ordenar un árbol?
  - ¿Cómo garantizar las condiciones de orden?
    - ¡Con invariantes!
  - *Binary Search Tree*, BST (árbol binario de búsqueda)

Buscando implementaciones logarítmicas

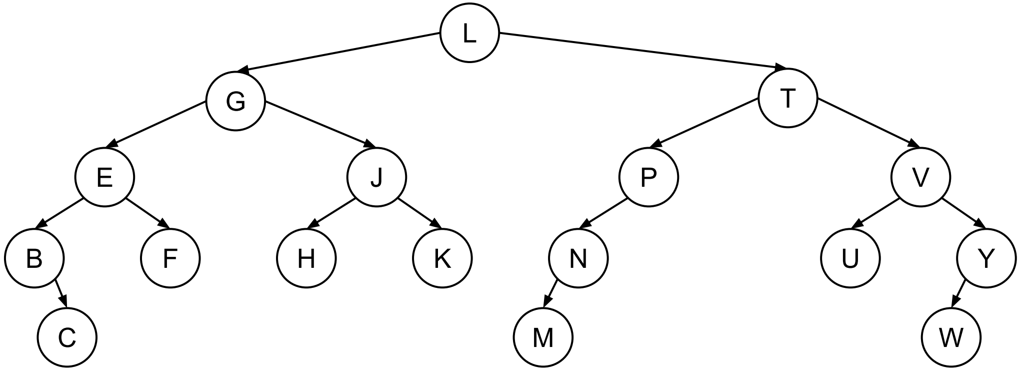
- ¿Podemos implementar **`Set`**s con costos logarítmicos?
- *Binary Search Tree*, BST (árbol binario de búsqueda)

```
data Tree a = EmptyT | NodeT a (Tree a) (Tree a)
```

- Invariante de BST: en **`(NodeT x ti td)`**
  - todos los elementos de **`ti`** son menores que **`x`**
  - todos los elementos de **`td`** son mayores que **`x`**
  - **`ti`** y **`td`** también cumplen el invariante de BST

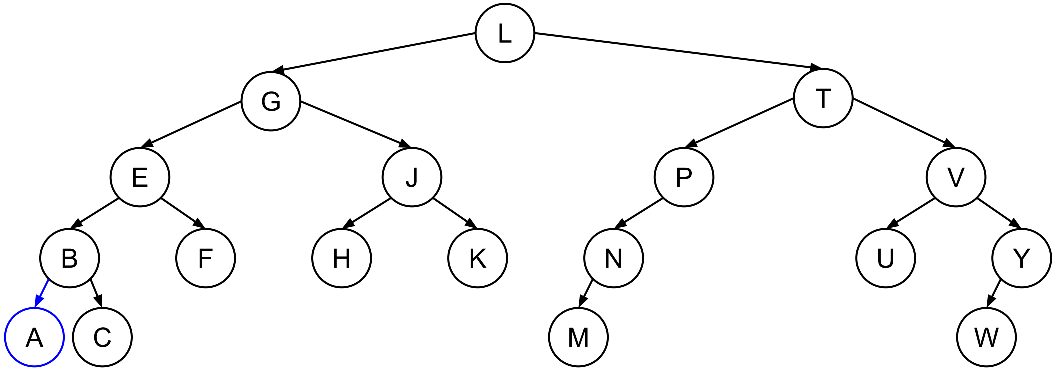
Buscando implementaciones logarítmicas

- *Binary Search Tree*, BST (árbol binario de búsqueda)
  - A la izquierda, menores, a la derecha, mayores.



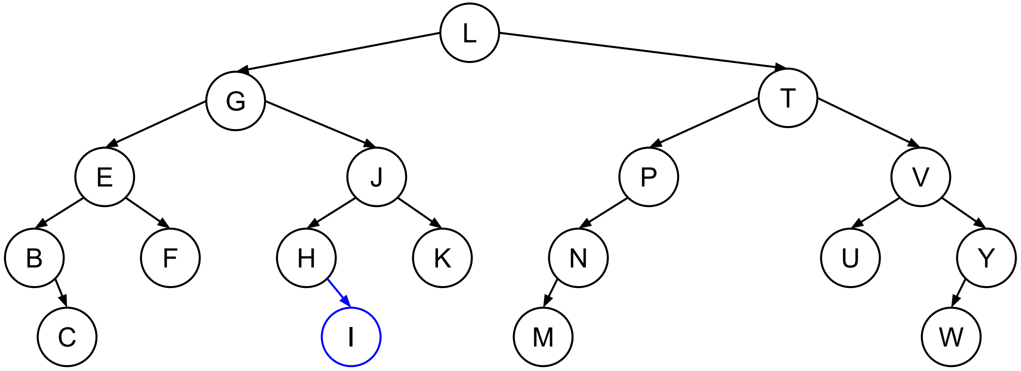
Buscando implementaciones logarítmicas

- *Binary Search Tree*, BST (árbol binario de búsqueda)
  - A la izquierda, menores, a la derecha, mayores.



Buscando implementaciones logarítmicas

- *Binary Search Tree*, BST (árbol binario de búsqueda)
  - A la izquierda, menores, a la derecha, mayores.



Buscando implementaci

```
data Set a

emptyS   :: Set a
addS     :: Ord a => a -> Set a -> Set a
belongs  :: Ord a => a -> Set a -> Bool
removeS  :: Ord a => a -> Set a -> Set a
set2list :: Ord a => Set a -> [a]
```

- ¿Podemos implementar **`Set`**s con costos logarítmicos?

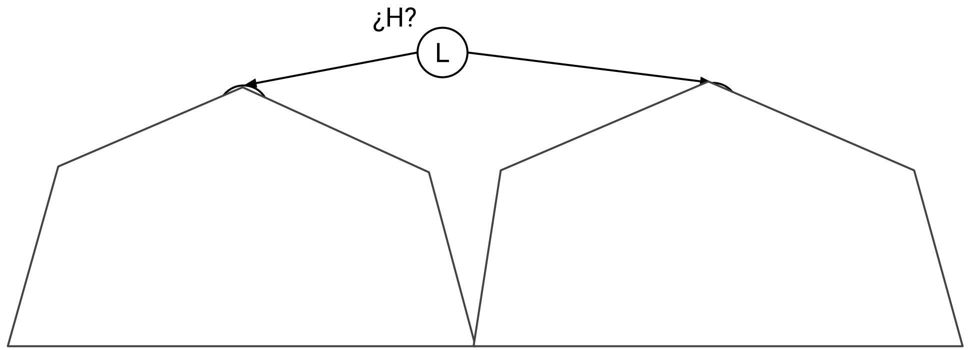
```
data Set a = S (Tree a)
  {- INV.REP.: en (S t), t cumple ser un BST -}
emptyS         = S EmptyT
set2list (S t) = inorder t
belongs …
addS …
removeS …
```

  - ¿Cómo hacer la búsqueda?
  - ¿Cómo insertar y borrar?

Buscando implementacion

```
data Set a = S (Tree a)
  {- INV.REP.: en (S t),
       * t cumple ser un BST -}
...
addS      :: Ord a => a -> Set a -> Set a
belongs   :: Ord a => a -> Set a -> Bool
removeS   :: Ord a => a -> Set a -> Set a
```

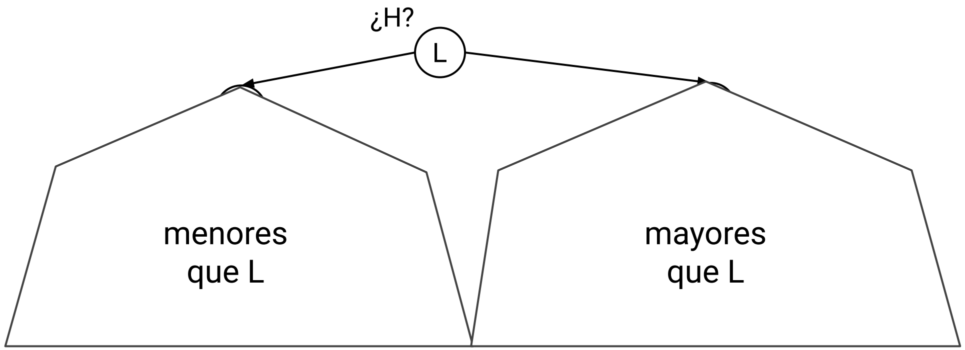
- ¿Cómo hacer la búsqueda en un BST?
  - Ejemplo: buscar si H está en el conjunto
    - ¿Hay que buscarlo en todos lados?
    - ¿Para qué lado estará? ¿De qué depende?



Buscando implementacion

```
data Set a = S (Tree a)
  {- INV.REP.: en (S t),
      * t cumple ser un BST -}
...
addS      :: Ord a => a -> Set a -> Set a
belongs   :: Ord a => a -> Set a -> Bool
removeS   :: Ord a => a -> Set a -> Set a
```

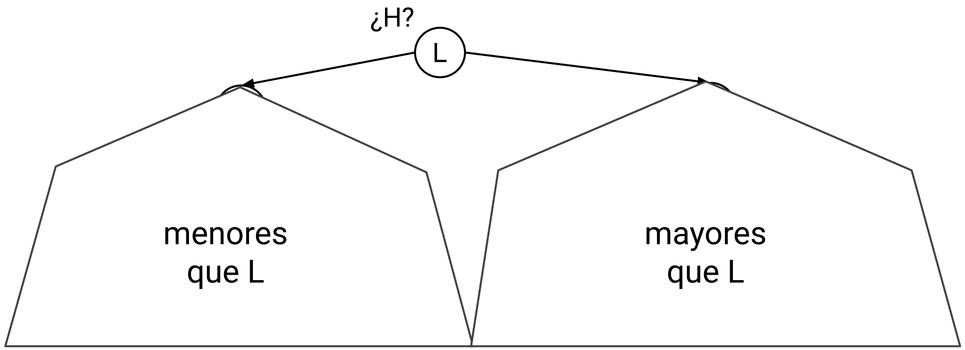
- ¿Cómo hacer la búsqueda en un BST?
  - Ejemplo: buscar si H está en el conjunto
    - ¿Hay que buscarlo en todos lados?
    - ¿Para qué lado estará? ¿De qué depende?



Buscando implementacion

```
data Set a = S (Tree a)
  {- INV.REP.: en (S t),
      * t cumple ser un BST -}
...
addS      :: Ord a => a -> Set a -> Set a
belongs   :: Ord a => a -> Set a -> Bool
removeS   :: Ord a => a -> Set a -> Set a
```

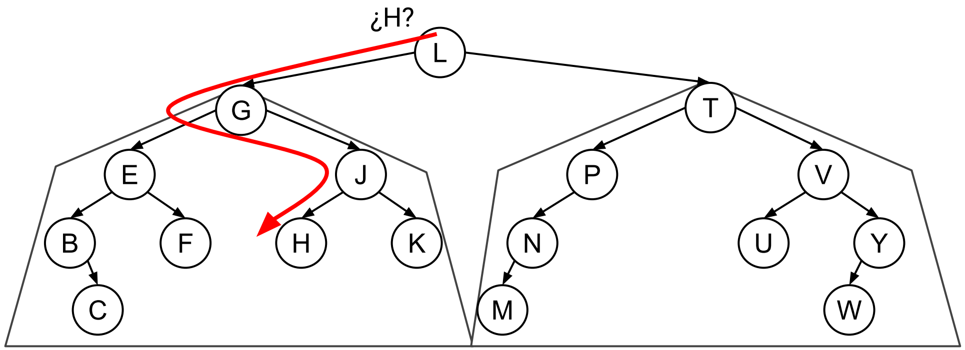
- ¿Cómo hacer la búsqueda en un BST?
  - Ejemplo: buscar si H está en el conjunto
    - Se compara el buscado con la raíz, y se decide para dónde ir
    - Luego se busca recursivamente



Buscando implementacion

```
data Set a = S (Tree a)
  {- INV.REP.: en (S t),
       * t cumple ser un BST -}
...
addS     :: Ord a => a -> Set a -> Set a
belongs  :: Ord a => a -> Set a -> Bool
removeS  :: Ord a => a -> Set a -> Set a
```

- ¿Cómo hacer la búsqueda en un BST?
  - Ejemplo: buscar si H está en el conjunto
    - Se compara el buscado con la raíz, y se decide para dónde ir
    - Luego se busca recursivamente



Buscando implementacion

```
data Set a = S (Tree a)
  {- INV.REP.: en (S t),
       * t cumple ser un BST -}
...
addS      :: Ord a => a -> Set a -> Set a
belongs   :: Ord a => a -> Set a -> Bool
removeS   :: Ord a => a -> Set a -> Set a
```

- ¿Cómo hacer la búsqueda en un BST?

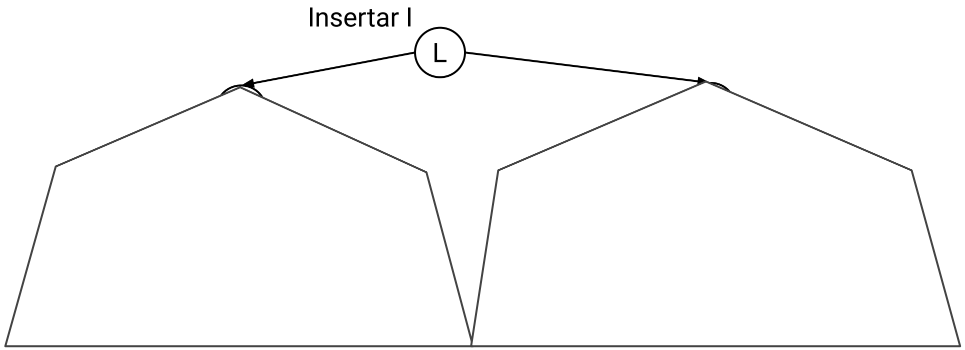
```
belongs x (S t) = buscarBST x t

buscarBST _ EmptyT           = False
buscarBST x (NodeT y ti td) =     -- PRECOND: t es BST
    if (x==y)        then True
     else if (x<y)  then buscarBST x ti
                     else buscarBST x td
```

- ¿Qué costo tiene?
  - Solamente recorre una rama
  - O(log n) (en un árbol balanceado)

Buscando implementacion

```
data Set a = S (Tree a)
  {- INV.REP.: en (S t),
      * t cumple ser un BST -}
...
addS      :: Ord a => a -> Set a -> Set a
belongs   :: Ord a => a -> Set a -> Bool
removeS   :: Ord a => a -> Set a -> Set a
```

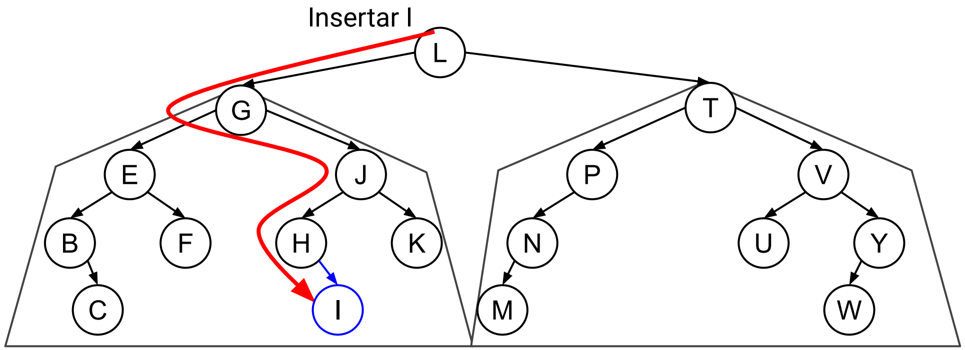
- ¿Cómo insertar en un BST?
  - Ejemplo: insertar l en el conjunto
    - ¿Hay que ponerlo en cualquier lado?
    - ¿De qué lado debe estar? ¿De qué depende?



Buscando implementacion

```
data Set a = S (Tree a)
  {- INV.REP.: en (S t),
       * t cumple ser un BST -}
...
addS      :: Ord a => a -> Set a -> Set a
belongs   :: Ord a => a -> Set a -> Bool
removeS   :: Ord a => a -> Set a -> Set a
```

- ¿Cómo insertar en un BST?
  - Ejemplo: buscar si H está en el conjunto
    - Se compara el buscado con la raíz, y se decide para dónde ir
    - Luego se busca recursivamente



Buscando implementacion

```
data Set a = S (Tree a)
  {- INV.REP.: en (S t),
       * t cumple ser un BST -}
...
addS      :: Ord a => a -> Set a -> Set a
belongs   :: Ord a => a -> Set a -> Bool
removeS   :: Ord a => a -> Set a -> Set a
```

- ¿Cómo insertar en un BST?

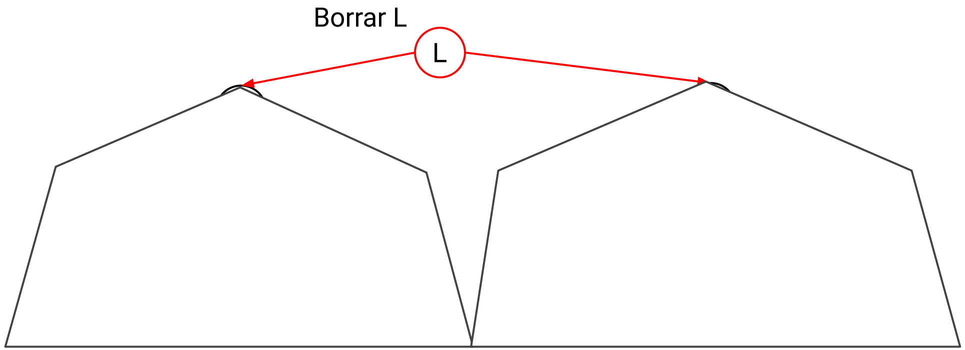
```
addS x (S t) = S (insertarBST x t)
insertarBST x EmptyT              = NodeT x EmptyT EmptyT
insertarBST x (NodeT y ti td) =          -- PRECOND: t es BST
    if (x==y)        then NodeT y ti td
     else if (x<y) then NodeT y (insertarBST x ti) td
                   else NodeT y ti (insertarBST x td)
```

- ¿Qué costo tiene?
  - Solamente recorre una rama
  - O(log n) (en un árbol balanceado)

Buscando implementacion

```
data Set a = S (Tree a)
  {- INV.REP.: en (S t),
       * t cumple ser un BST -}
...
addS      :: Ord a => a -> Set a -> Set a
belongs   :: Ord a => a -> Set a -> Bool
removeS   :: Ord a => a -> Set a -> Set a
```

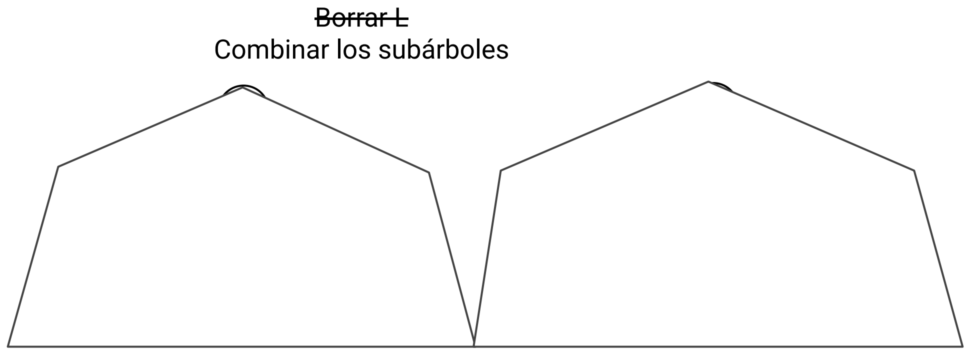
- ¿Cómo borrar en un BST?
  - Ejemplo: borrar L del conjunto
    - Primero hay que encontrarlo (como en buscar)
    - ¡Al borrarlo, quedan 2 subárboles! ¿Cómo combinarlos?



Buscando implementacion

```
data Set a = S (Tree a)
  {- INV.REP.: en (S t),
       * t cumple ser un BST -}
...
addS      :: Ord a => a -> Set a -> Set a
belongs   :: Ord a => a -> Set a -> Bool
removeS   :: Ord a => a -> Set a -> Set a
```

- ¿Cómo borrar en un BST?
  - Ejemplo: borrar L del conjunto
    - Primero hay que encontrarlo (como en buscar)
    - ¡Al borrarlo, quedan 2 subárboles! ¿Cómo combinarlos?



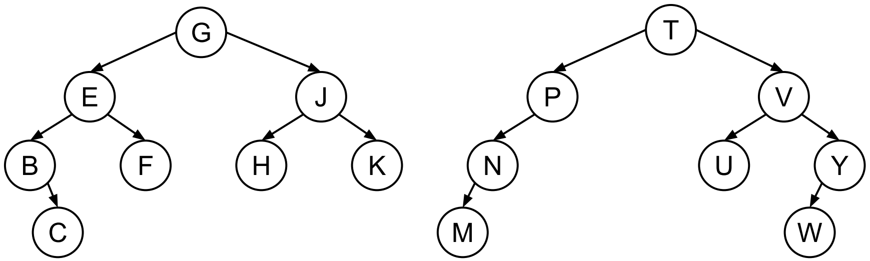
Buscando implementacion

```
data Set a = S (Tree a)
  {- INV.REP.: en (S t),
       * t cumple ser un BST -}
...
addS      :: Ord a => a -> Set a -> Set a
belongs   :: Ord a => a -> Set a -> Bool
removeS   :: Ord a => a -> Set a -> Set a
```

- ¿Cómo borrar en un BST?
  - … ¿Cómo combinar los subárboles?
  - Debe quedar un BST. Se necesita alguien para la raíz…
  - ¿Quiénes pueden ser los candidatos?

~~Borrar L~~

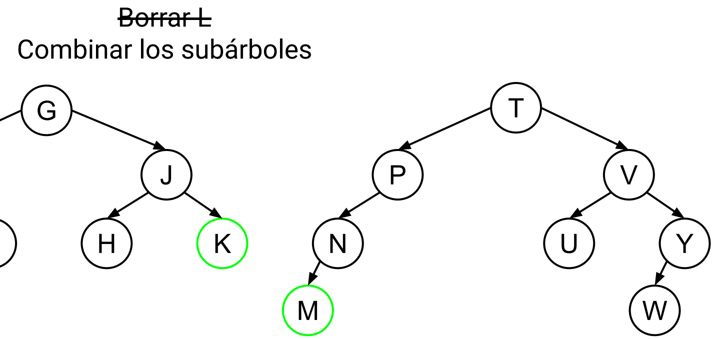
Combinar los subárboles



Buscando implementacion

```
data Set a = S (Tree a)
  {- INV.REP.: en (S t),
      * t cumple ser un BST -}
...
addS      :: Ord a => a -> Set a -> Set a
belongs   :: Ord a => a -> Set a -> Bool
removeS   :: Ord a => a -> Set a -> Set a
```

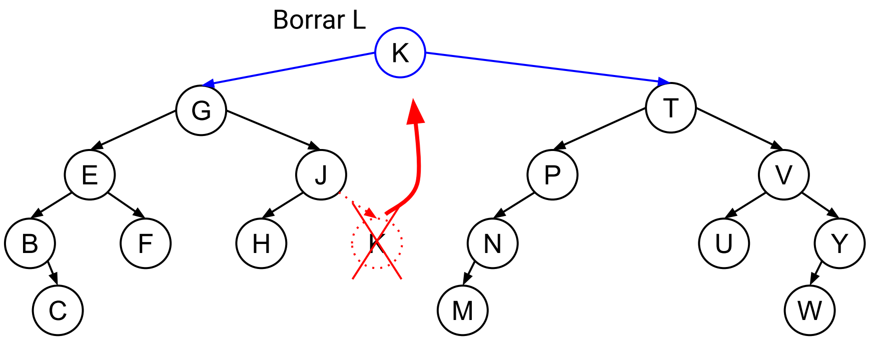
- ¿Cómo borrar en un BST?
  - … ¿Cómo combinar los subárboles?
  - Debe quedar un BST. Se necesita alguien para la raíz…
  - ¿Quiénes pueden ser los candidatos?



Buscando implementacion

```
data Set a = S (Tree a)
  {- INV.REP.: en (S t),
       * t cumple ser un BST -}
...
addS      :: Ord a => a -> Set a -> Set a
belongs   :: Ord a => a -> Set a -> Bool
removeS   :: Ord a => a -> Set a -> Set a
```

- ¿Cómo borrar en un BST?
  - … ¿Quiénes pueden ser los candidatos?
  - El mínimo de la derecha, o el máximo de la izquierda
  - Se lo busca y se lo lleva a la raíz



Buscando implementacion

```
data Set a = S (Tree a)
  {- INV.REP.: en (S t),
       * t cumple ser un BST -}
...
addS      :: Ord a => a -> Set a -> Set a
belongs   :: Ord a => a -> Set a -> Bool
removeS   :: Ord a => a -> Set a -> Set a
```

- ¿Cómo borrar en un BST?

```
removeS x (S t) = S (borrarBST x t)

borrarBST _ EmptyT           = EmptyT
borrarBST x (NodeT y ti td) =          -- PRECOND: t es BST
    if (x==y)        then rearmarBST ti td
     else if (x<y) then NodeT y (borrarBST x ti) td
                    else NodeT y ti (borrarBST x td)
```

- ¿Qué costo tiene?
  - Solamente recorre una rama: O(log n) (en un árbol balanceado)
  - Falta hacer **rearmarBST** en costo O(log n)

Buscando implementacion

```
data Set a = S (Tree a)
  {- INV.REP.: en (S t),
       * t cumple ser un BST -}
...
addS       :: Ord a => a -> Set a -> Set a
belongs    :: Ord a => a -> Set a -> Bool
removeS    :: Ord a => a -> Set a -> Set a
```

- ¿Cómo borrar en un BST?

```
rearmarBST :: Ord a => Tree a -> Tree a -> Tree a
  -- PRECOND: ambos árboles son BSTs

rearmarBST EmptyT td = td
rearmarBST ti     td = NodeT (maxBST ti) (delMaxBST ti) td

maxBST (NodeT x _ EmptyT) = x         -- PRECOND: no es vacío
maxBST (NodeT _ _ td)     = maxBST td

delMaxBST (NodeT _ ti EmptyT) = ti   -- PRECOND: no es vacío
delMaxBST (NodeT x ti td)     = NodeT x ti (delMaxBST td)
```

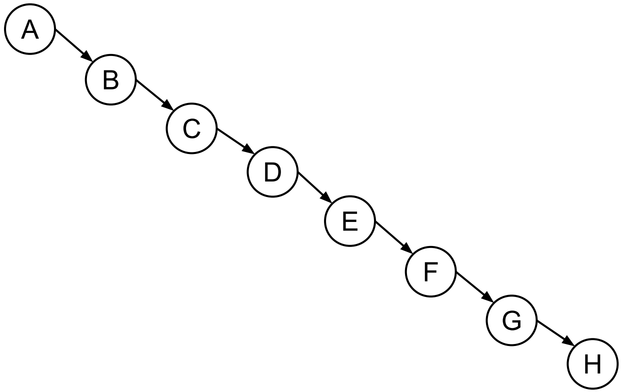
  - Alternativamente, se puede usar **`splitMaxBST`**

Buscando implementaciones logarítmicas

- ¿Podemos implementar `Set`s con costos logarítmicos?
  - Todas las implementaciones vistas
    - solamente recorre una rama
    - Tienen costo O(log n) en un árbol balanceado
    - Pero, ¿cuál es el peor caso?

Buscando implementaciones logarítmicas

- ¿Podemos implementar `Sets` con costos logarítmicos?
  - Todas las implementaciones vistas
    - solamente recorre una rama
    - Tienen costo O(log n) en un árbol balanceado
    - Pero, ¿cuál es el peor caso?
      - Insertar elementos en orden…



Buscando implementaciones logarítmicas

- ¿Podemos implementar `Sets` con costos logarítmicos?
  - Todas las implementaciones vistas
    - solamente recorre una rama
    - Tienen costo O(log n) en un árbol balanceado
    - Pero, ¿cuál es el peor caso?
      - Insertar elementos en orden:
        TODOS los hijos izquierdos son vacíos
      - La rama más larga es O(n)
    - ¿Cómo asegurar que el árbol se mantiene balanceado?

Árboles balanceados: AVLs

Implementaciones logarítmicas

- Para mantener el balance, se precisa otro invariante
- Hay muchas formas posibles de lograrlo
  - *AVLs* (Adelson-Velsky & Landis)
  - Red-Black Trees
  - …
- Invariante de AVL: en `(NodeT x ti td)`
  - la diferencia de alturas entre `ti` y `td` es menor o igual a 1
  - `ti` y `td` también cumplen el invariante de AVLs

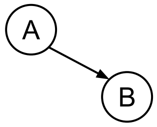
Implementaciones logarítmicas

- Invariante de AVL: en **`(NodeT x ti td)`**
  - la diferencia de alturas entre **`ti`** y **`td`** es menor o igual a 1
  - **`ti`** y **`td`** también cumplen el invariante de AVLs

Es AVL

Implementaciones logarítmicas

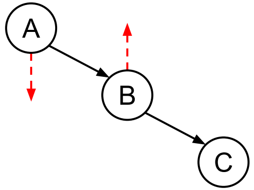
- Invariante de AVL: en `(NodeT x ti td)`
  - la diferencia de alturas entre `ti` y `td` es menor o igual a 1
  - `ti` y `td` también cumplen el invariante de AVLs



Es AVL

Implementaciones logarítmicas

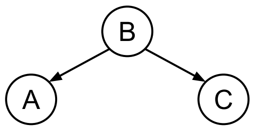
- Invariante de AVL: en **`(NodeT x ti td)`**
  - la diferencia de alturas entre **`ti`** y **`td`** es menor o igual a 1
  - **`ti`** y **`td`** también cumplen el invariante de AVLs



NO es AVL

Implementaciones logarítmicas

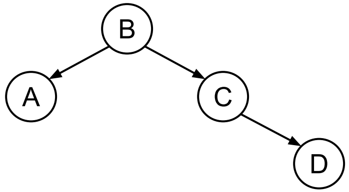
- Invariante de AVL: en **`(NodeT x ti td)`**
  - la diferencia de alturas entre **`ti`** y **`td`** es menor o igual a 1
  - **`ti`** y **`td`** también cumplen el invariante de AVLs



Es AVL

Implementaciones logarítmicas

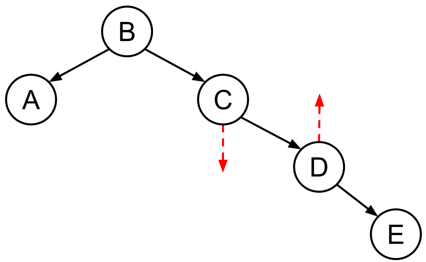
- Invariante de AVL: en **`(NodeT x ti td)`**
  - la diferencia de alturas entre **`ti`** y **`td`** es menor o igual a 1
  - **`ti`** y **`td`** también cumplen el invariante de AVLs



Es AVL

Implementaciones logarítmicas

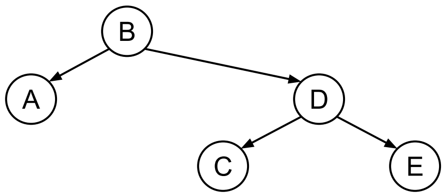
- Invariante de AVL: en **`(NodeT x ti td)`**
  - la diferencia de alturas entre **`ti`** y **`td`** es menor o igual a 1
  - **`ti`** y **`td`** también cumplen el invariante de AVLs



NO es AVL

Implementaciones logarítmicas

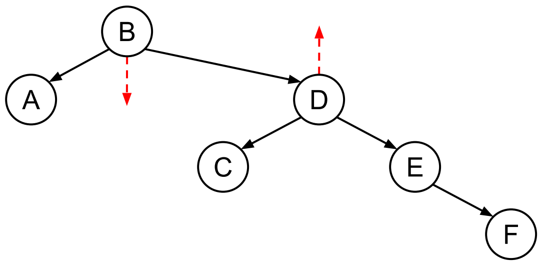
- Invariante de AVL: en **`(NodeT x ti td)`**
  - la diferencia de alturas entre **`ti`** y **`td`** es menor o igual a 1
  - **`ti`** y **`td`** también cumplen el invariante de AVLs



Es AVL

Implementaciones logarítmicas

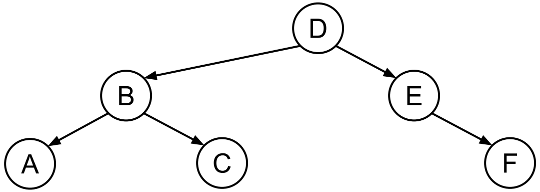
- Invariante de AVL: en **`(NodeT x ti td)`**
  - la diferencia de alturas entre **`ti`** y **`td`** es menor o igual a 1
  - **`ti`** y **`td`** también cumplen el invariante de AVLs



NO es AVL

Implementaciones logarítmicas

- Invariante de AVL: en **`(NodeT x ti td)`**
  - la diferencia de alturas entre **`ti`** y **`td`** es menor o igual a 1
  - **`ti`** y **`td`** también cumplen el invariante de AVLs



Es AVL

Implementaciones logarítmicas

- Invariante de AVL: en **`(NodeT x ti td)`**
  - la diferencia de alturas entre **`ti`** y **`td`** es menor o igual a 1
  - **`ti`** y **`td`** también cumplen el invariante de AVLs
- Para implementarlo eficientemente, debe mantenerse la altura

```
data AVL a = EmptyAVL | NodeAVL Int a (AVL a) (AVL a)
  {- INV.REP.: en NodeAVL h x ti td
       * h es la altura del árbol
       * la diferencia de alturas de ti y td es <= 1
       * ti y tf son AVLs                           -}
heightAVL :: AVL a -> Int       -- O(1)
heightAVL EmptyAVL           = 0
heightAVL (NodeAVL h _ _ _) = h
```

Implementaciones logarítmicas

- Invariante de AVL: en `(NodeAVL h x ti td)`
  - la diferencia de alturas entre `ti` y `td` es menor o igual a 1
  - `ti` y `td` también cumplen el invariante de AVLs
  - `h` es la altura del árbol
- ¿Cómo garantizar el invariante de AVL?
  - Al insertar y al borrar
  - Se requiere ver cuáles lugares pueden alterar eso
    - Donde se reconstruyen AVLs…
    - Cada `NodeT` (ahora `NodeAVL`) debe revisarse

Implementaciones logarít

```
data Set a = S (AVL a)
  {- INV.REP.: en (S t),
      * t cumple ser un BST y un AVL    -}
data AVL a = EmptyAVL
           | NodeAVL Int a (AVL a) (AVL a)
...
```

- ¿Cómo insertar en un AVL?

```
addS x (S t) = S (insertAVL x t)

insertAVL x EmptyAVL           = armarAVL x EmptyAVL EmptyAVL
insertAVL x (NodeAVL _ y ti td) =
    if (x==y)       then armarAVL y ti td
     else if (x<y) then armarAVL y (insertAVL x ti) td
                   else armarAVL y ti (insertAVL x td)
```

- ¿Qué costo tiene?
  - O(log n) en peor caso (¡el árbol es balanceado!)
  - La operación **armarAVL** debe tomar AVLs y devolver AVLs

Implementaciones logarít

```
data Set a = S (AVL a)
  {- INV.REP.: en (S t),
       * t cumple ser un BST y un AVL    -}
data AVL a = EmptyAVL
           | NodeAVL Int a (AVL a) (AVL a)
...
```

- ¿Cómo borrar en un AVL?

```
removeS x (S t) = S (deleteAVL x t)

deleteAVL _ EmptyAVL                = EmptyAVL
deleteAVL x (NodeAVL _ y ti td) =
     if (x==y)        then rearmarAVL ti td
      else if (x<y) then armarAVL y (deleteAVL x ti) td
                         else armarAVL y ti (deleteAVL x td)
```

- ¿Qué costo tiene?
  - O(log n) (¡el árbol es balanceado!)
  - La operación **`armarAVL`** debe tomar AVLs y devolver AVLs

```
data Set a = S (AVL a)
  {- INV.REP.: en (S t),
       * t cumple ser un BST y un AVL    -}
data AVL a = EmptyAVL
             | NodeAVL Int a (AVL a) (AVL a)
...
```

Implementaciones logarít

- ¿Cómo borrar en un AVL?

```
rearmarAVL EmptyAVL td = td
rearmarAVL ti       td = armarAVL (maxAVL ti)
                                  (delMaxAVL ti) td

maxAVL (NodeAVL _ x _ EmptyAVL) = x
maxAVL (NodeAVL _ _ _ td)       = maxAVL td

delMaxAVL (NodeAVL _ _ ti EmptyT) = ti
delMaxAVL (NodeAVL _ x ti td)     =
                    armarAVL x ti (delMaxAVL td)
```

- La operación **`armarAVL`** debe tomar AVLs y devolver AVLs

Implementaciones logarít

```
data Set a = S (AVL a)
  {- INV.REP.: en (S t),
       * t cumple ser un BST y un AVL    -}
data AVL a = EmptyAVL
           | NodeAVL Int a (AVL a) (AVL a)
...
```

- ¿Cómo mantener el invariante en AVLs?
  - La operación **`armarAVL`** debe tomar AVLs y devolver AVLs
  - Debe analizar los casos, y rebalancear o “rotar” (este código excede el alcance de la materia)

```
armarAVL x ti td =                                     -- O(1)
  let hi = heightAVL ti  -- O(1)
      hd = heightAVL td  -- O(1)
   in if abs (hi-hd) <= 1   then symAVL   x ti td  -- Arma sin rotar
       else if hi == hd + 2 then leftAVL  x ti td  -- Rota a izquierda
       else if hd == hi + 2 then rightAVL x ti td  -- Rota a derecha
       else                      error "Se viola el invariante!"
                                 -- Otros casos que nunca se alcanzan
```

Implementaciones logarít

```
data Set a = S (AVL a)
  {- INV.REP.: en (S t),
       * t cumple ser un BST y un AVL    -}
data AVL a = EmptyAVL
           | NodeAVL Int a (AVL a) (AVL a)
...
```

- ¿Cómo mantener el invariante en AVLs?
  - Armado de AVL sin rotar
    (este código excede el alcance de la materia)

```
symAVL x ti td = NodeAVL (newHeight ti td) x ti td  -- O(1)
newHeightAVL ti td = 1 + max (heightAVL ti)
                             (heightAVL td)
```

```
data Set a = S (AVL a)
  {- INV.REP.: en (S t),
       * t cumple ser un BST y un AVL    -}
data AVL a = EmptyAVL
             | NodeAVL Int a (AVL a) (AVL a)
...
```

Implementaciones logarít

- ¿Cómo mantener el invariante en AVLs?
  - Rotación a izquierda (excede el alcance de la materia)

Simple

```
        x                 xi
       / \               /  \
      /   \             /    \
    xi    td  --->   tii      x
   / \                       / \
  / ti\                     /   \
tii   tid                 tid   td
```

Doble

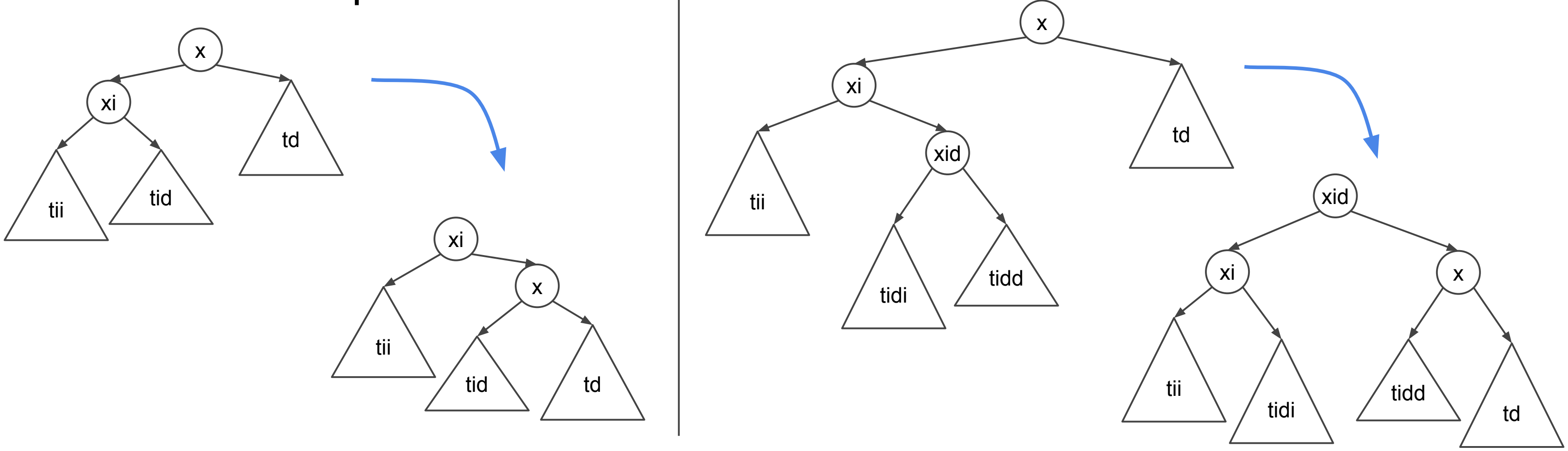
```
        x                       xid
       / \                     /   \
      /   \                   /     \
    xi    td  --->          xi       x
   / \                     / \      / \
  / ti\                   /   \    /   \
tii   xid               tii  tidi tidd  td
     /   \
    / tid \
  tidi   tidd
```

Implementaciones logarít

```
data Set a = S (AVL a)
  {- INV.REP.: en (S t),
       * t cumple ser un BST y un AVL    -}
data AVL a = EmptyAVL
           | NodeAVL Int a (AVL a) (AVL a)
...
```

- ¿Cómo mantener el invariante en AVLs?
  - Rotación a izquierda (excede el alcance de la materia)

Simple | Doble



Implementaciones logarít

```
data Set a = S (AVL a)
  {- INV.REP.: en (S t),
       * t cumple ser un BST y un AVL    -}
data AVL a = EmptyAVL
           | NodeAVL Int a (AVL a) (AVL a)
...
```

- ¿Cómo mantener el invariante en AVLs?
  - Rotación a izquierda

    (este código excede el alcance de la materia)

```
leftAVL x (NodeAVL hi xi tii tid) td =    -- O(1)
   let hii = heightAVL tii
       hid = heightAVL tid
    in if hii >= hid
        then symAVL xi tii (symAVL x tid td)       -- Simple
        else let (NodeAVL _ xid tidi tidd) = tid  -- Doble
               in symAVL xid (symAVL xi tii  tidi)
                             (symAVL x  tidd td)
```

Implementaciones logarít

```
data Set a = S (AVL a)
  {- INV.REP.: en (S t),
       * t cumple ser un BST y un AVL    -}
data AVL a = EmptyAVL
           | NodeAVL Int a (AVL a) (AVL a)
...
```

- ¿Cómo mantener el invariante en AVLs?
  - Rotación a derecha
    (este código excede el alcance de la materia)

```
rightAVL x ti (NodeAVL hd xd tdi tdd) =   -- O(1)
   let hdi = heightAVL tdi
       hdd = heightAVL tdd
    in if hdi <= hdd
        then symAVL xd (symAVL x ti tdi) tdd       -- Simple
        else let (NodeAVL _ xdi tdii tdid) = tdi   -- Doble
              in symAVL xdi (symAVL x  ti   tdii)
                            (symAVL xd tdid tdd)
```

Implementaciones logarítmicas

- La altura de un AVL se mantiene logarítmica
  - Las operaciones son O(log n) en peor caso
  - Los invariantes ayudan a diseñar el código correcto
- Otros invariantes de balanceo son posibles
  - Red-Black trees
  - AA trees (Arne Andersson trees)
  - Tango trees
  - y muchas otras variantes

Nuevas implementaciones: Heaps

Implementaciones logarítmicas

- ¿Podemos implementar **`PQ`**s con costos logarítmicos?

```
data PriorityQueue a

emptyPQ      ...
isEmptyPQ    ...
insertPQ     ...
findMinPQ    ...
deleteMinPQ  ...
```

  - ¿Cómo encontrar el mínimo?
  - ¿Cómo insertar y borrar?

Implementaciones logarítmicas

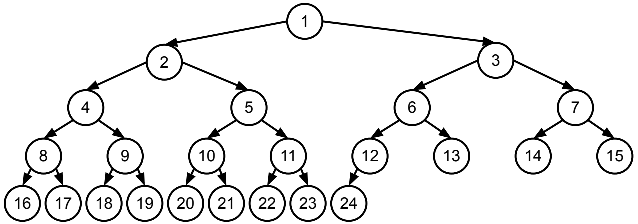
- ¿Costo logarítmico en el caso de priority queues?
  - Los BSTs podrían servir, pero no serían 100% adecuados
  - Debemos usar árboles para tener costo logarítmico…
  - Pero se precisa algún invariante más preciso
  - ¿Qué debería valer para que findMinPQ sea O(1)?
    - Algún otro invariante…
    - Da lugar a los llamados *heaps* (“*jips*”, montículos, o pilones)

Implementaciones logarítmicas

- Invariante de ***heap*** (*“jip”*, montículo, pilón):
  - La raíz es el mínimo de todos los elementos
  - Los subárboles cumplen el invariante *heap*
- Restricciones adicionales llevan a casos específicos
  - Ej: agregando “árbol lleno” se obtienen los *heaps binarios*
  - Otras restricciones llevan a otras formas de heaps
    - Binomial heaps
    - Skewed heaps
    - 2-3 heaps
    - Leftists heaps
    - Factorial heaps
    - y muchas otras

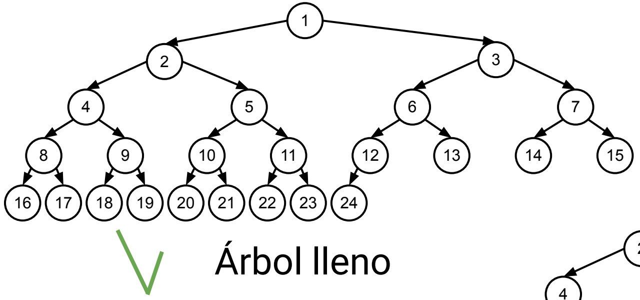
Heaps binarias

- Invariante de ***heap*** (montículo, pilón):
  - La raíz es el mínimo de todos los elementos
  - Los subárboles cumplen el invariante *heap*
- Invariante de ***árbol lleno*** (*full tree*):
  - Todos los niveles, salvo quizás el último están completos
  - El último no tiene “agujeros” de izquierda a derecha

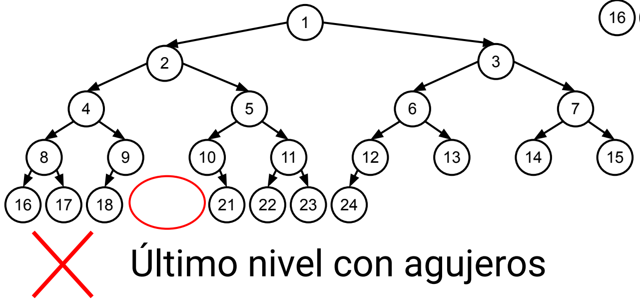


Heaps binarias

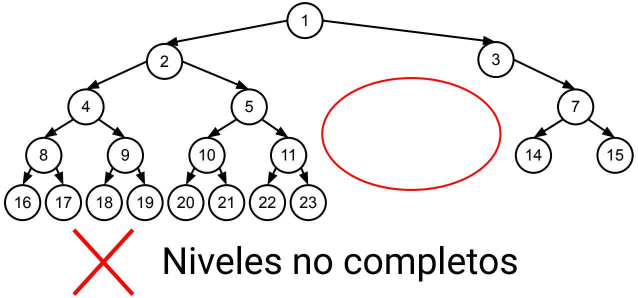
- Invariante de ***árbol lleno*** (*full tree*):
  - Todos los niveles, salvo quizás el último están completos
  - El último no tiene “agujeros” de izquierda a derecha



Árbol lleno



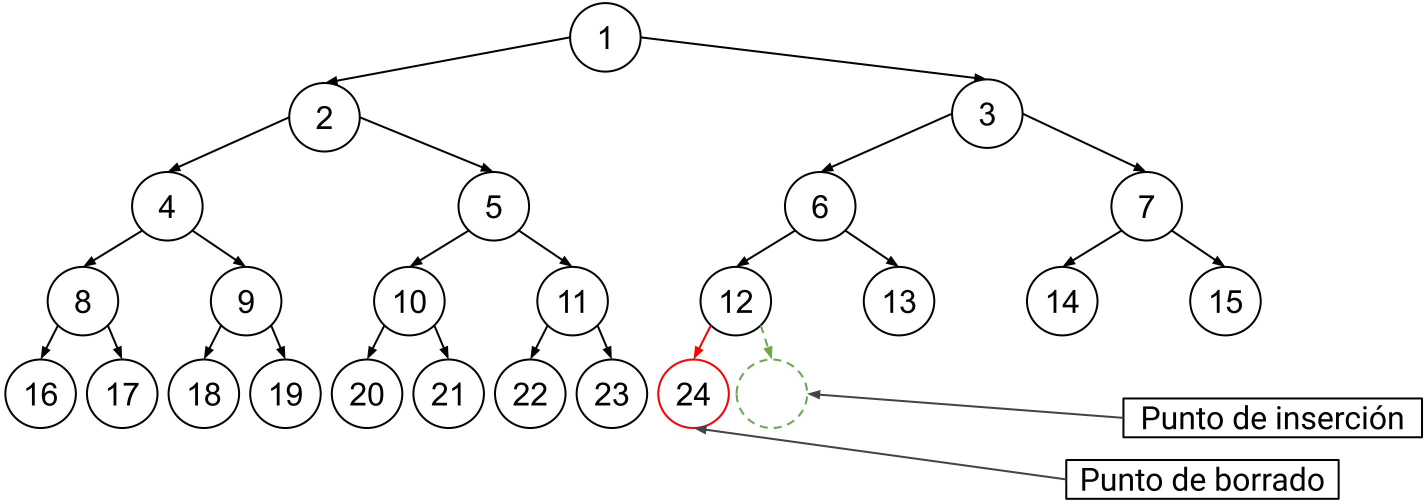
Último nivel con agujeros



Niveles no completos

Heaps binarias

- Ventajas de un árbol lleno
  - tiene un único lugar donde insertar y donde borrar
  - al usar memoria, se puede implementar muy eficiente



Heaps binarias

- Ventajas de un árbol lleno
  - tiene un único lugar donde insertar y donde borrar
  - al usar memoria, se puede implementar muy eficiente
- Desventajas en el modelo declarativo
  - se precisan datos adicionales para ubicar el lugar de inserción/borrado
- Al combinar ambos invariantes, se pueden deducir las operaciones de inserción y borrado

Implementaciones logar

```
data PriorityQueue a

emptyPQ      :: PriorityQueue a
isEmptyPQ    :: PriorityQueue a -> Bool
insertPQ     :: Ord a => a -> PriorityQueue a
                               -> PriorityQueue a
findMinPQ    :: Ord a => PriorityQueue a -> a
deleteMinPQ  :: Ord a => PriorityQueue a
                           -> PriorityQueue a
```

- ¿Podemos implementar PQ

```
data Dir = Izq | Der

data PriorityQueue a = PQ [Dir] (Tree a)
  {- INV.REP.: en (PQ pos t)
        * t cumple ser un heap
        * t cumple ser un árbol lleno
        * pos indica el camino hasta la posición
             de inserción en t                    -}
```

  - ¿Cómo encontrar el mínimo?
  - ¿Cómo insertar y borrar?

Implementaciones logar

```
data Dir = Izq | Der

data PriorityQueue a = PQ [Dir] (Tree a)
{- INV.REP.: en (PQ pos t)
       * t cumple ser un heap
       * t cumple ser un árbol lleno
       * pos indica el camino hasta la
            posición de inserción en t   -}
```

- ¿Podemos implementar **PQ**s con costos logarítmicos?

```
emptyPQ             = PQ [] EmptyT        -- O(1)
isEmptyPQ (PQ _ t) = isEmptyT t         -- O(1)
findMinPQ (PQ _ t) = root t             -- O(1)
                       -- Por Inv.Rep. es el mínimo
root (NodeT x _ _) = x                  -- O(1)
```

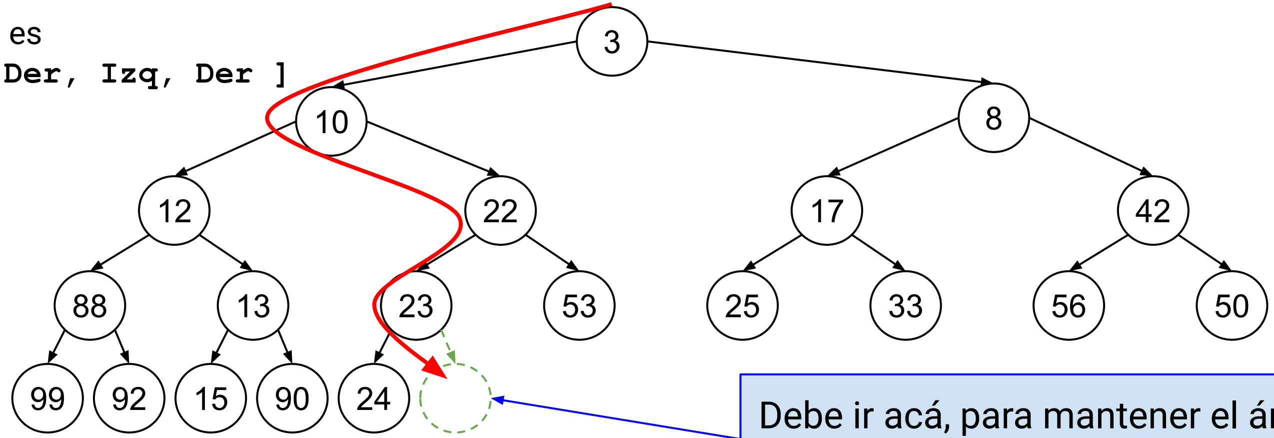
  - ¿Cómo encontrar el mínimo?
    - ¡La raíz del árbol!

Implementaciones logar

```
data Dir = Izq | Der

data PriorityQueue a = PQ [Dir] (Tree a)
{- INV.REP.: en (PQ pos t)
        * t cumple ser un heap
        * t cumple ser un árbol lleno
        * pos indica el camino hasta la
             posición de inserción en t   -}
```

- ¿Podemos implementar **PQ**s con costos logarítmicos?
  - ¿Cómo insertar? Deben preservarse los invariantes
    - Ej: insertar el 5

La posición es
`[ Izq, Der, Izq, Der ]`

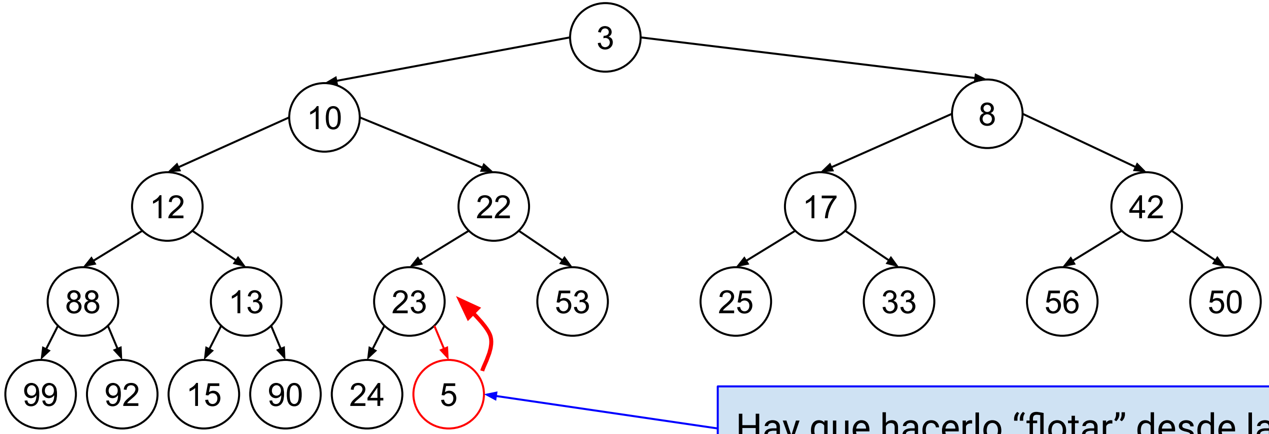


Debe ir acá, para mantener el árbol lleno.
¡Pero rompe el invariante de heap!

Implementaciones logar

```
data Dir = Izq | Der

data PriorityQueue a = PQ [Dir] (Tree a)
{- INV.REP.: en (PQ pos t)
       * t cumple ser un heap
       * t cumple ser un árbol lleno
       * pos indica el camino hasta la
           posición de inserción en t   -}
```

- ¿Podemos implementar **`PQ`**s con costos logarítmicos?
  - ¿Cómo insertar? Deben preservarse los invariantes
    - Ej: insertar el 5

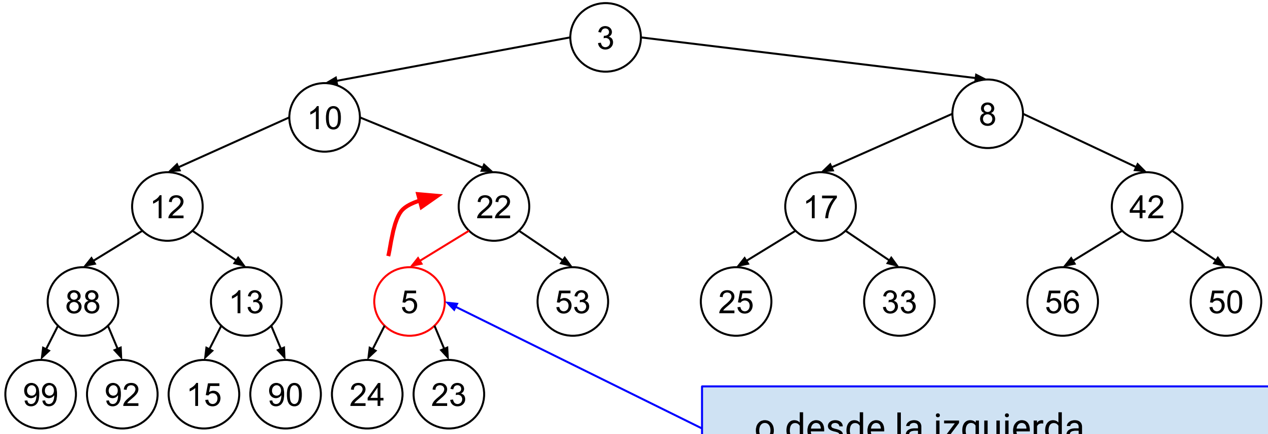


Hay que hacerlo “flotar” desde la derecha…

Implementaciones logar

```
data Dir = Izq | Der

data PriorityQueue a = PQ [Dir] (Tree a)
{- INV.REP.: en (PQ pos t)
       * t cumple ser un heap
       * t cumple ser un árbol lleno
       * pos indica el camino hasta la
           posición de inserción en t   -}
```

- ¿Podemos implementar **`PQ`**s con costos logarítmicos?
  - ¿Cómo insertar? Deben preservarse los invariantes
    - Ej: insertar el 5

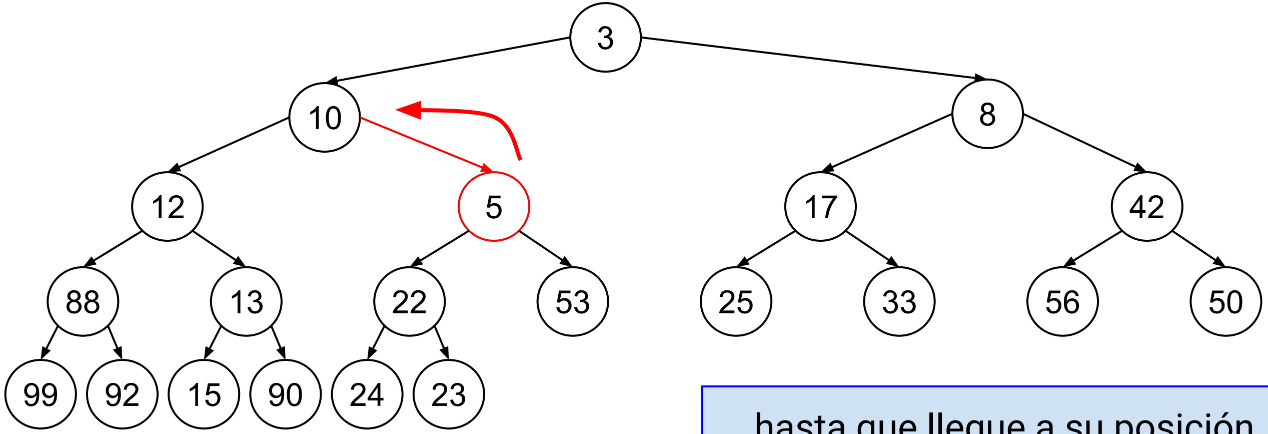


… o desde la izquierda…

Implementaciones logar

```
data Dir = Izq | Der

data PriorityQueue a = PQ [Dir] (Tree a)
{- INV.REP.: en (PQ pos t)
       * t cumple ser un heap
       * t cumple ser un árbol lleno
       * pos indica el camino hasta la
            posición de inserción en t   -}
```

- ¿Podemos implementar **PQ**s con costos logarítmicos?
  - ¿Cómo insertar? Deben preservarse los invariantes
    - Ej: insertar el 5

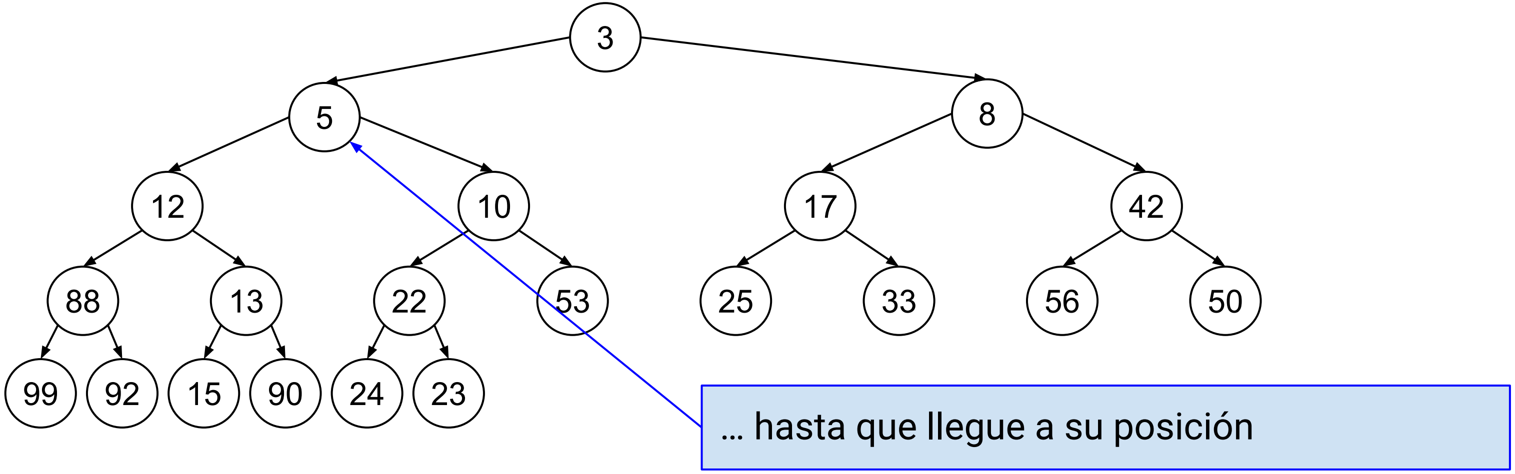


… hasta que llegue a su posición

Implementaciones logar

```
data Dir = Izq | Der

data PriorityQueue a = PQ [Dir] (Tree a)
{- INV.REP.: en (PQ pos t)
       * t cumple ser un heap
       * t cumple ser un árbol lleno
       * pos indica el camino hasta la
            posición de inserción en t   -}
```

- ¿Podemos implementar **`PQ`**s con costos logarítmicos?
  - ¿Cómo insertar? Deben preservarse los invariantes
    - Ej: insertar el 5



Implementaciones logar

```
data Dir = Izq | Der

data PriorityQueue a = PQ [Dir] (Tree a)
{- INV.REP.: en (PQ pos t)
        * t cumple ser un heap
        * t cumple ser un árbol lleno
        * pos indica el camino hasta la
           posición de inserción en t   -}
```

- ¿Podemos implementar **PQ**s con costos logarítmicos?
  - ¿Cómo insertar? Deben preservarse los invariantes
    - Insertar y flotar, y actualizar la posición

```
insertPQ x (PQ pos t) = PQ (nextPos pos)
                           (insertIn (reverse pos) x t)
insertIn []         x EmptyT           =    -- O(log n)
                      NodeT x EmptyT EmptyT
insertIn (Izq:pos) x (NodeT m ti td) =
                      flotarIzq m (insertIn pos x ti) td
insertIn (Der:pos) x (NodeT m ti td) =
                      flotarDer m ti (insertIn pos x td)
```

Implementaciones logar

```
data Dir = Izq | Der

data PriorityQueue a = PQ [Dir] (Tree a)
{- INV.REP.: en (PQ pos t)
       * t cumple ser un heap
       * t cumple ser un árbol lleno
       * pos indica el camino hasta la
            posición de inserción en t   -}
```

- ¿Podemos implementar **PQ**s con costos logarítmicos?
  - ¿Cómo insertar? Deben preservarse los invariantes
    - Flotar lleva un elemento hacia arriba

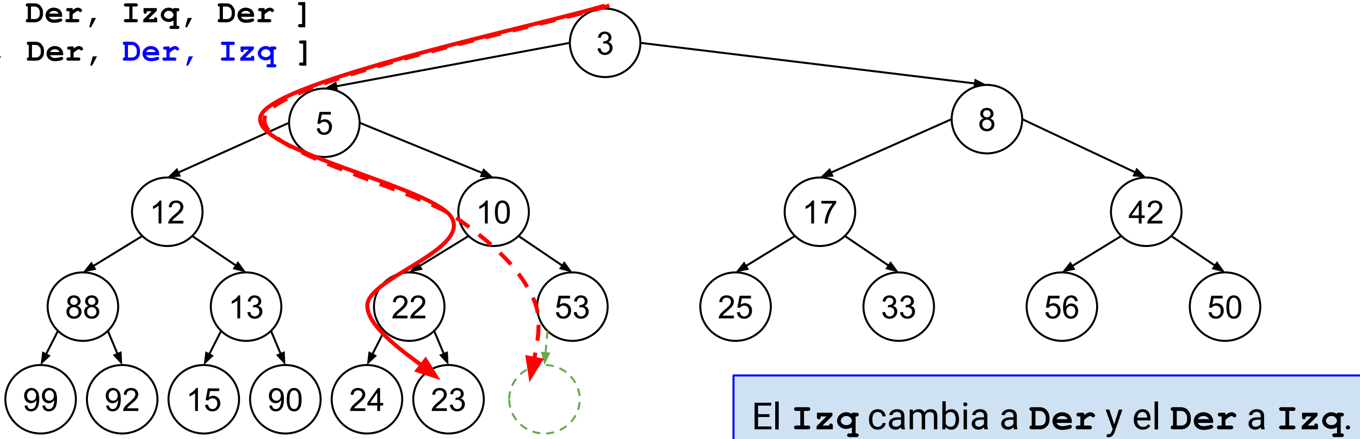
```
flotarIzq m (NodeT m' tii tid) td =        -- O(1)
  {- PRECOND: los argumentos son heaps -}
    if m <= m' then NodeT m  (NodeT m' tii tid) td
               else NodeT m' (NodeT m  tii tid) td

flotarDer m tii (NodeT m' tdi tdd) =       -- O(1)
  {- PRECOND: los argumentos son heaps -}
    if m <= m' then NodeT m  ti (NodeT m' tdi tdd)
               else NodeT m' ti (NodeT m  tdi tdd)
```

Implementaciones logar

```
data Dir = Izq | Der

data PriorityQueue a = PQ [Dir] (Tree a)
{- INV.REP.: en (PQ pos t)
        * t cumple ser un heap
        * t cumple ser un árbol lleno
        * pos indica el camino hasta la
           posición de inserción en t   -}
```

- ¿Podemos implementar **PQ**s con costos logarítmicos?
  - ¿Cómo insertar? Deben preservarse los invariantes
    - La posición debe actualizarse

Cambia `[ Izq, Der, Izq, Der ]`
por `[ Izq, Der, Der, Izq ]`



El **Izq** cambia a **Der** y el **Der** a **Izq**.
Si cambia un **Der**, sigue hacia arriba.

Implementaciones logar

```
data Dir = Izq | Der

data PriorityQueue a = PQ [Dir] (Tree a)
{- INV.REP.: en (PQ pos t)
       * t cumple ser un heap
       * t cumple ser un árbol lleno
       * pos indica el camino hasta la
            posición de inserción en t   -}
```

- ¿Podemos implementar **PQ**s con costos logarítmicos?
  - ¿Cómo insertar? Deben preservarse los invariantes
    - La posición se guarda invertida por eficiencia

```
nextPos []        = [Izq]
nextPos (Izq:pos) = Der : pos
nextPos (Der:pos) = Izq : nextPos pos
```

    - Es como incrementar en uno una representación binaria…

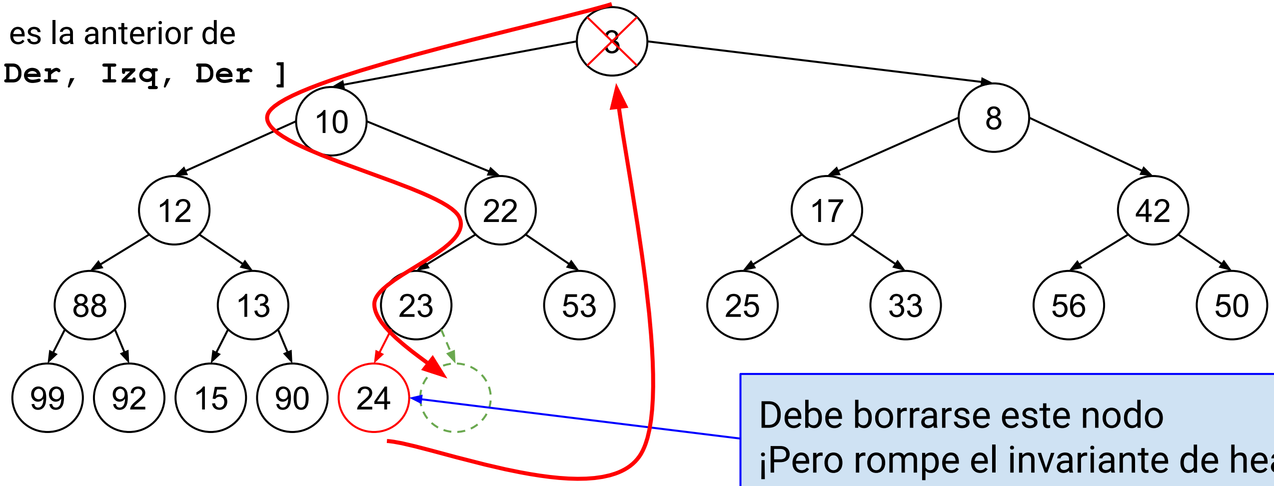
El **Izq** cambia a **Der** y el **Der** a **Izq**.
Si cambia un **Der**, sigue hacia arriba.

Implementaciones logar

```
data Dir = Izq | Der

data PriorityQueue a = PQ [Dir] (Tree a)
{- INV.REP.: en (PQ pos t)
       * t cumple ser un heap
       * t cumple ser un árbol lleno
       * pos indica el camino hasta la
          posición de inserción en t   -}
```

- ¿Podemos implementar **`PQ`**s con costos logarítmicos?
  - ¿Cómo borrar? Deben preservarse los invariantes
    - Hay una única posición de borrado…

La posición es la anterior de
`[ Izq, Der, Izq, Der ]`

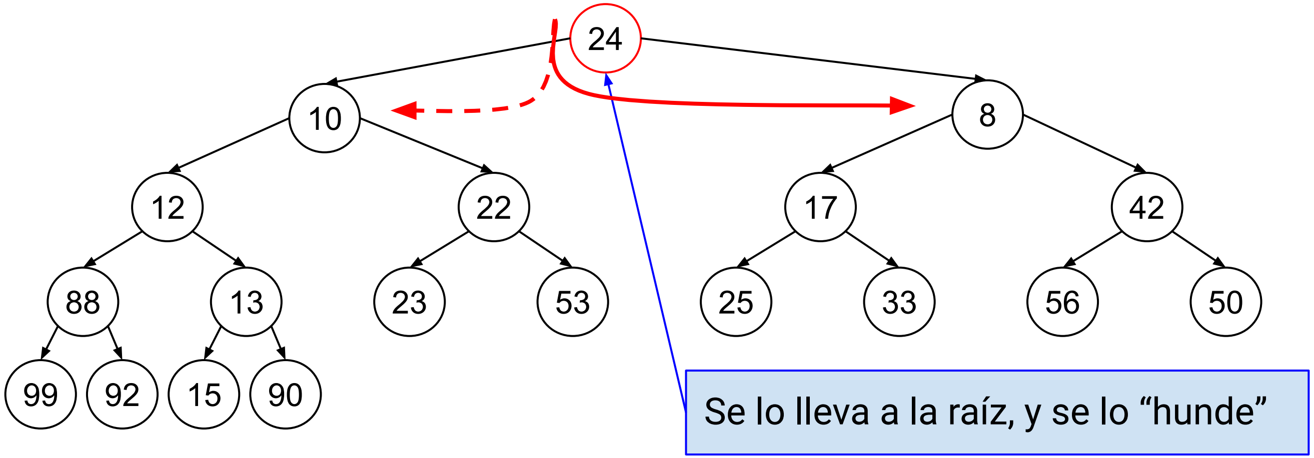


Debe borrarse este nodo
¡Pero rompe el invariante de heap!

Implementaciones logar

```
data Dir = Izq | Der

data PriorityQueue a = PQ [Dir] (Tree a)
{- INV.REP.: en (PQ pos t)
        * t cumple ser un heap
        * t cumple ser un árbol lleno
        * pos indica el camino hasta la
             posición de inserción en t   -}
```

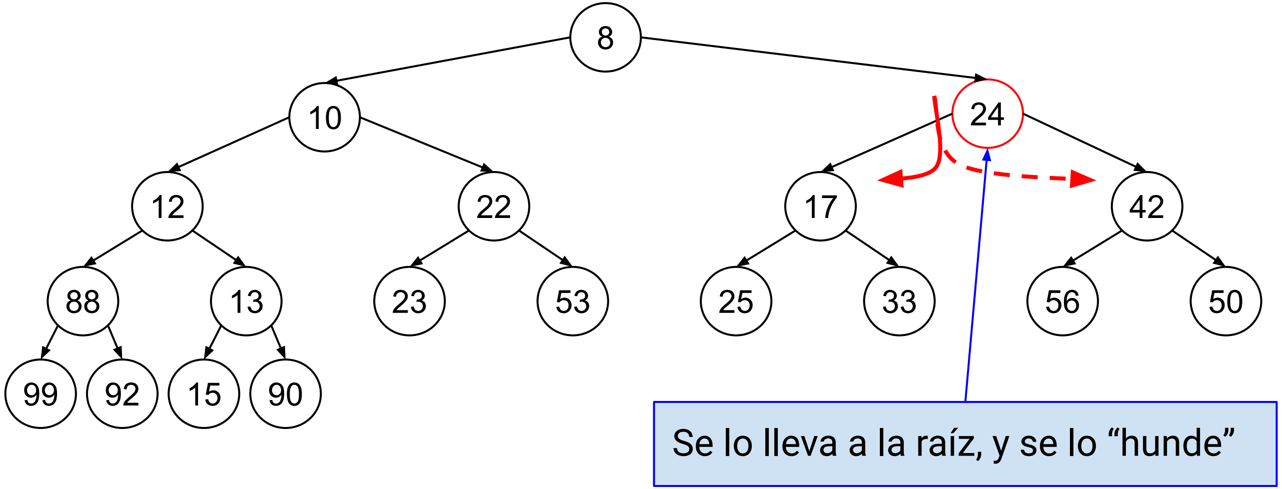
- ¿Podemos implementar **PQ**s con costos logarítmicos?
  - ¿Cómo borrar? Deben preservarse los invariantes
    - Hay una única posición de borrado…



Implementaciones logar

```
data Dir = Izq | Der

data PriorityQueue a = PQ [Dir] (Tree a)
{- INV.REP.: en (PQ pos t)
        * t cumple ser un heap
        * t cumple ser un árbol lleno
        * pos indica el camino hasta la
            posición de inserción en t   -}
```

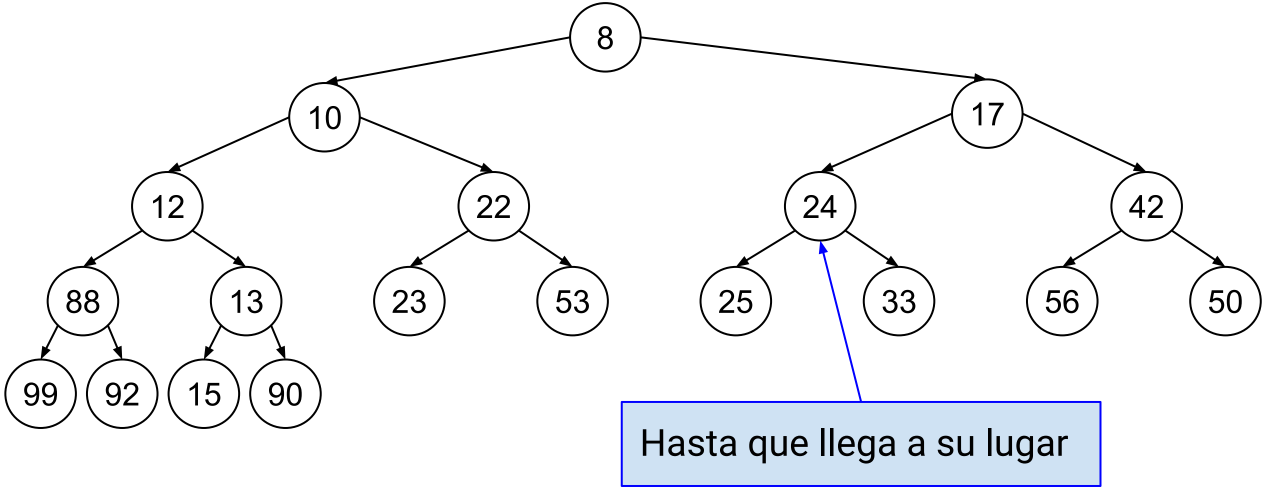
- ¿Podemos implementar **`PQ`**s con costos logarítmicos?
  - ¿Cómo borrar? Deben preservarse los invariantes
    - Hay una única posición de borrado…



Implementaciones logar

```
data Dir = Izq | Der

data PriorityQueue a = PQ [Dir] (Tree a)
{- INV.REP.: en (PQ pos t)
       * t cumple ser un heap
       * t cumple ser un árbol lleno
       * pos indica el camino hasta la
            posición de inserción en t   -}
```

- ¿Podemos implementar **`PQ`**s con costos logarítmicos?
  - ¿Cómo borrar? Deben preservarse los invariantes
    - Hay una única posición de borrado…



Implementaciones logar

```
data Dir = Izq | Der

data PriorityQueue a = PQ [Dir] (Tree a)
{- INV.REP.: en (PQ pos t)
       * t cumple ser un heap
       * t cumple ser un árbol lleno
       * pos indica el camino hasta la
           posición de inserción en t   -}
```

- ¿Podemos implementar **PQ**s con costos logarítmicos?
  - ¿Cómo borrar? Deben preservarse los invariantes
    - Sacar y hundir, y actualizar la posición

```
deleteMinPQ x (PQ pos t) =
              PQ preP (deleteIn (reverse preP) x t)
  where preP = prevPos pos

deleteIn []  (NodeT _ EmptyT EmptyT) = EmptyT
deleteIn pos t                       =
  let (m', NodeT _ ti' td') = splitAt pos t
   in hundir m' ti' td'
```

Implementaciones loga[...]

```
data Dir = Izq | Der

data PriorityQueue a = PQ [Dir] (Tree a)
{- INV.REP.: en (PQ pos t)
       * t cumple ser un heap
       * t cumple ser un árbol lleno
       * pos indica el camino hasta la
            posición de inserción en t   -}
```

- ¿Podemos implementar **PQ**s con costos logarítmicos?
  - ¿Cómo borrar? Deben preservarse los invariantes
    - Hundir lleva un elemento hacia abajo

```
{- PRECOND: * los 3 elementos forman un árbol lleno
            * los subárboles son heaps               -}
hundir m Empty              EmptyT            = NodeT m EmptyT EmptyT
hundir m (NodeT mi tii tid) EmptyT            =
  if m <= mi then NodeT m  (NodeT  mi tii tid) EmptyT
             else NodeT mi (hundir m  tii tid) EmptyT
hundir m (NodeT mi tii tid)  (NodeT md tdi tdd) =
  if m <= mi && m <= md
   then NodeT m  (NodeT mi tii tid)  (NodeT md tdi tdd)
   else if mi <= md
          then NodeT mi (hundir m tii tid)  (NodeT md tdi tdd)
          else NodeT md (NodeT mi tii tid)  (hundir m tdi tdd)
```

```
data Dir = Izq | Der

data PriorityQueue a = PQ [Dir] (Tree a)
{- INV.REP.: en (PQ pos t)
       * t cumple ser un heap
       * t cumple ser un árbol lleno
       * pos indica el camino hasta la
            posición de inserción en t   -}
```

Implementaciones logar

- ¿Podemos implementar **PQ**s con costos logarítmicos?
  - ¿Cómo borrar? Deben preservarse los invariantes
    - Separar devuelve el elemento en la pos dada y el árbol sin él

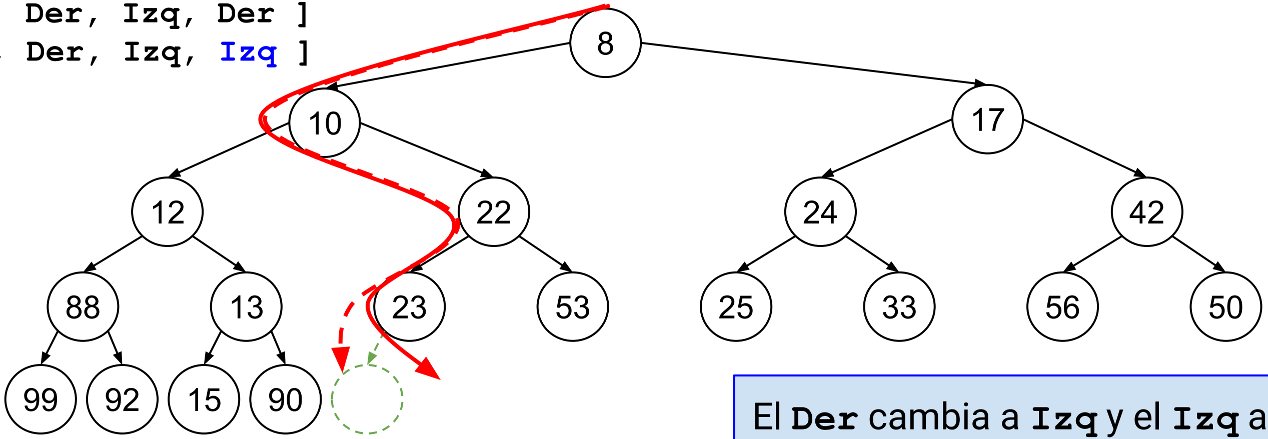
```
splitAt []        (NodeT m EmptyT EmptyT) = (m, EmptyT)
splitAt (Izq:pos) (NodeT m ti td)         =
                       let (m', ti') = splitAt pos ti
                        in (m', NodeT m ti' td)
splitAt (Der:pos) (NodeT m ti td)         =
                       let (m', td') = splitAt pos td
                        in (m', NodeT m ti td')
```

Implementaciones logar

```
data Dir = Izq | Der

data PriorityQueue a = PQ [Dir] (Tree a)
{- INV.REP.: en (PQ pos t)
        * t cumple ser un heap
        * t cumple ser un árbol lleno
        * pos indica el camino hasta la
            posición de inserción en t   -}
```

- ¿Podemos implementar **`PQ`**s con costos logarítmicos?
  - ¿Cómo insertar? Deben preservarse los invariantes
    - La posición debe actualizarse

Cambia `[ Izq, Der, Izq, Der ]`
por `[ Izq, Der, Izq, Izq ]`



El **`Der`** cambia a **`Izq`** y el **`Izq`** a **`Der`**.
Si cambia un **`Izq`**, sigue hacia arraba.

Implementaciones logar

```
data Dir = Izq | Der

data PriorityQueue a = PQ [Dir] (Tree a)
{- INV.REP.: en (PQ pos t)
       * t cumple ser un heap
       * t cumple ser un árbol lleno
       * pos indica el camino hasta la
            posición de inserción en t   -}
```

- ¿Podemos implementar **`PQ`**s con costos logarítmicos?
  - ¿Cómo insertar? Deben preservarse los invariantes
    - La posición se guarda invertida por eficiencia

```
prevPos [Izq]       = []
prevPos (Der:pos) = Izq : pos
prevPos (Izq:pos) = Der : prevPos pos
```

    - Es como decrementar en uno una representación binaria…

El **`Der`** cambia a **`Izq`** y el **`Izq`** a **`Der`**.
Si cambia un **`Izq`**, sigue hacia arraba.

Observaciones

- Los árboles BST, AVL y los Heaps
  - NO son Tipos Abstractos por sí mismos
  - Son formas de implementar otros TADs
  - Requieren conocer detalles de cómo hacerlos bien
  - En algunos casos se pueden mejorar al usar memoria
- Al programar profesionalmente
  - Todos estas implementaciones existen
  - Hay más, en muchos casos mejores para ciertos usos
  - Pero conocer cómo lograrlas es importante

Resumen

Resumen

- Nuevas categorías para medir eficiencia
  - Costo logarítmico
  - Costo “eneloguene”
- Se pueden usar árboles para mejorar eficiencia
  - Debe buscarse que estén balanceados
  - Muchos costos mejoran de O(n) a O(log n)
  - Los invariantes son cruciales para saber qué hacer
  - Deben conocerse detalles para poder implementar bien