PROGRAMACIÓN FUNCIONAL

Tipos de Datos: Algoritmos sobre Árboles (1)

Algoritmos sobre árboles

- Maps y sus implementaciones
  - Definición
  - Implementación trivial
  - Árboles de búsqueda (BSTs)
  - Árboles AVL

Mappings

- Tipo abstracto `Map` (mapping, o asociación)
  - acceso a valores (`v`) indexados por claves (`k`)
  - las claves deben poder compararse con (==)
  - se utilizan módulos para abstraer las distintas implementaciones
  - algunas implementaciones pueden requerir que las claves sean ordenables con (<), etc.

Mappings

- Tipo abstracto Map: interfaz

```
module MappingInterface where

  class Map map where

    emptyM :: map k v
    lookupM :: Ord k => map k v -> k -> Maybe v
    addM    :: Ord k => map k v -> k -> v -> map k v
          -- Sobreescribe valores anteriores
    deleteM :: Ord k => map k v -> k -> map k v
    domM    :: Eq k => map k v -> [k]  -- Cada clave, una vez

          ⋮
```

Mappings

- Usos posibles

```
type Agenda map      = map Nombre Telefono
type Diccionario map = map Palabra [Significado]
type Tesauro map     = map Palabra [Palabra]
type Memoria map     = map Variable Valor
type Array map       = map Int Valor
```

  - ¡Los arrays son formas de map con claves numéricas, implementados en memoria para acceso en orden constante ($\mathcal{O}(1)$)!
  - Lenguajes como Python los proveen como tipo de datos primitivo (implementados con hashing)

Mappings (listas)

- Implementación trivial (basada en listas)

```
data MapL k v = M [ (k, v) ]
  -- INVARIANTE DE REPRESENTACIÓN:
  --    * no hay claves repetidas en la lista
instance Map MapL where
  emptyM                = M [ ]
  lookupM (M kvs) k     = lookup kvs k
  addM (M kvs) k v      = M (modificar kvs k v)
  deleteM (M kvs) k     = M (borrar kvs k)
  domM (M kvs)          = mapFst kvs
    ⋮
```

Mappings (listas)

- Implementación trivial (auxiliares)

```
    ⋮
modificar [ ]             k v = [ (k,v) ]
modificar ((k',v'):kvs) k v =
          if k==k'  then (k',v) : kvs
                    else (k',v') : modificar kvs k v

lookup [ ]              k = Nothing
lookup ((k',v'):kvs) k =
          if k==k'  then Just v
                    else lookup kvs k
    ⋮
```

Mappings (listas)

- Implementación trivial (auxiliares)

```
  ⋮
borrar [ ]              k = [ ]
borrar ((k',v'):kvs) k =
            if k==k'  then kvs
                      else (k',v') : borrar kvs k
```

  - ¡Las 3 son $\mathcal{O}(n)$! En peor caso y en promedio
  - Ordenar las claves no mejora el orden

Mappings (BSTs)

- Implementaciones con árboles (de búsqueda)

```
        ⋮
type MapB k v = BST k v
data BST k v = EmptyT | NodeT k v (BST k v) (BST k v)
  {- INVARIANTE DE REPRESENTACIÓN:
       * las claves se pueden comparar (con (>))
       * las claves del hijo izquierdo son menores que la raíz
       * las claves del hijo derecho son mayores a la raíz
       * ambos hijos son BSTs
  -}
        ⋮
```

Mappings (BSTs)

- Implementaciones con BSTs (cont.)

```
        ⋮
instance Map MapB where
  emptyM = EmptyT
  domM = listarClaves

-- auxiliar para domM (fuera del instance)
listarClaves EmptyT = [ ]
listarClaves (NodeT k v t1 t2) =   (listarClaves z f t1)
        ⋮                           ++ [k] ++
                                   (listarClaves z f t2)
```

Mappings (BSTs)

- Implementaciones con BSTs (alternativa)

⋮

```
domM = foldT [ ] (\k v ks1 ks2 -> ks1++[k]++ks2)

-- auxiliar para domM (fuera del instance)
foldT z f EmptyT = z
foldT z f (NodeT k v t1 t2) = f k v (foldT z f t1)
                                    (foldT z f t2)
```

⋮

Mappings (BSTs)

- Implementaciones con BSTs (cont.)

```
  ⋮

addM EmptyT                 k v =
        NodeT k v EmptyT EmptyT

addM (NodeT k' v' t1 t2) k v =
        if   k==k'    then NodeT k' v t1 t2
        else if k<k' then NodeT k' v' (addM t1 k v) t2
        else                NodeT k' v' t1 (addM t2 k v)

  ⋮
```

Mappings (BSTs)

- Implementaciones con BSTs (cont.)

```
    ⋮

deleteM EmptyT  k v = EmptyT

deleteM (NodeT k' v' t1 t2) k v =
            if   k==k'    then rearmarBSTcon t1 t2
            else if k<k' then NodeT k' v' (deleteM t1 k v) t2
            else                  NodeT k' v' t1 (deleteM t2 k v)

    ⋮
```

Mappings (BSTs)

- Implementaciones con BSTs (auxiliares)

```
      ⋮
-- auxiliares para deleteM (fuera del instance)
rearmarBSTcon EmptyT t2 = t2
rearmarBSTcon t1 t2 = let (y, t1') = splitMax t1
                      in NodeT y t1' t2

splitMax (NodeT x t1 EmptyT) = (x, t1)
splitMax (NodeT x t1 t2) = let (y, t2') = splitMax t2
                           in (y, NodeT x t1 t2')
      ⋮
```

Mappings (BSTs)

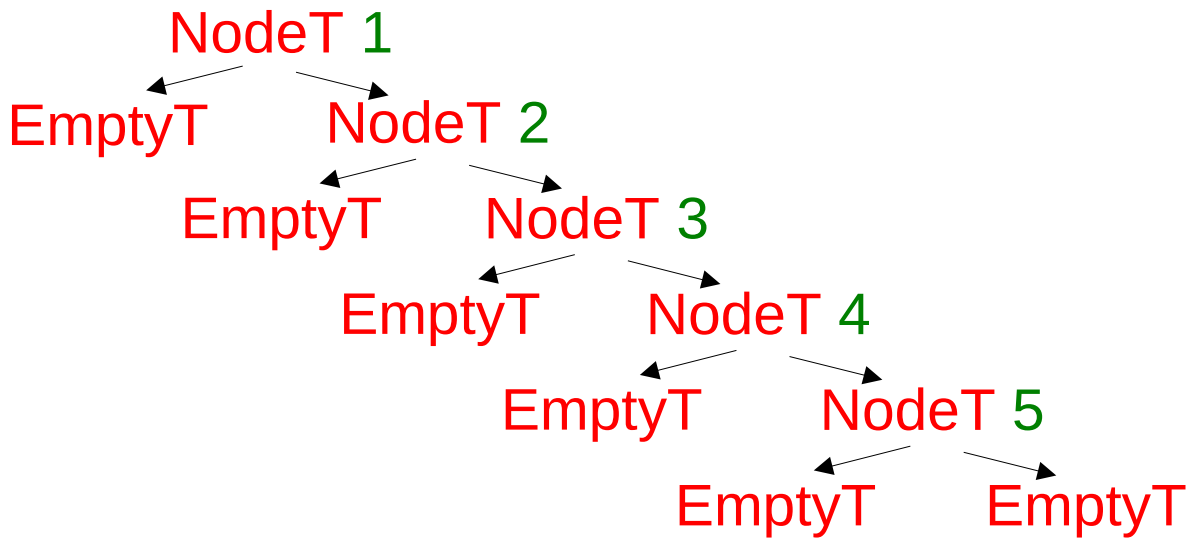
- Implementaciones con BSTs (de búsqueda)

```
  ⋮
lookupM EmptyT                 k v = Nothing
lookupM (NodeT k' v' t1 t2) k v =
          if   k==k'    then Just v
          else if k<k' then lookupM t1 k v
          else                lookupM t2 k v
```

  - La búsqueda, la inserción y el borrado
    - son logarítmicas $\mathcal{O}(\log n)$ en promedio
    - pero lineales $\mathcal{O}(n)$ en peor caso (¿por qué?)

Mappings (BSTs)

- ¡Un árbol desbalanceado es el peor caso!



  - La operaciones son lineales $\mathcal{O}(n)$ en este tipo de árboles
  - Se requieren árboles balanceados

Mappings (AVLs)

- ¿Cómo conseguir balanceo?
  - Precisamos un invariante que lo garantice...
  - Los AVL son una forma de garantizar balanceo
    - Piden que
      - la diferencia de altura de los subárboles no sea mayor a 1,
      - y que los subárboles sean AVLs.
    - Para realizar la comparación, deben almacenar la altura junto con los datos (para no recomputar)
  - Hay otras formas (RedBlack Trees, etc.)

Mappings (AVLs)

- Implementaciones con árboles
  (de búsqueda balanceados)

```
        ⋮
type MapA k v = AVL k v
data AVL k v = E | N Int k v (AVL k v) (AVL k v)

      {- Invariante de representación
         * son BSTs (como antes)
`        * son AVLs
             - la diferencia de altura entre los hijos es de a lo sumo 1
             - los hijos son AVLs
             - el entero es la altura del árbol
       -}
        ⋮
```

Mappings (AVLs)

- Implementaciones con AVLs (cont.)

```
        ⋮
instance Map MapA where
  emptyM = E
  domM = foldAVL [ ] (\h k v ks1 ks2 -> ks1++[k]++ks2)

-- auxiliar para domM (fuera del instance)
foldAVL z f E = z
foldAVL z f (N h k v t1 t2) = f h k v  (foldAVL z f t1)
                                       (foldAVL z f t2)
        ⋮
```

Mappings (BSTs)

- Implementaciones con AVLs (cont.)

```
  ⋮
addM E                    k v = armarAVL k v E E
addM (N h k' v' t1 t2) k v =
    if   k==k'     then  armarAVL k' v t1 t2
    else if k<k'   then  armarAVL k' v' (addM t1 k v) t2
    else                 armarAVL k' v' t1 (addM t2 k v)

  ⋮
```

  - el código rojo es **la única** diferencia con los BSTs

Mappings (BSTs)

- Implementaciones con AVLs (cont.)

```
    ⋮
deleteM E  k v = E
deleteM (N h k' v' t1 t2) k v =
    if   k==k'     then rearmarAVLcon t1 t2
    else if k<k' then armarAVL k' v' (deleteM t1 k v) t2
    else                armarAVL k' v' t1 (deleteM t2 k v)


    ⋮
```

Mappings (BSTs)

- Implementaciones con AVLs (auxiliares)

```
      ⋮
-- auxiliares para deleteM (fuera del instance)
rearmarAVLcon E t2 = t2
rearmarAVLcon t1 t2 = let (k,v,t1') = splitMaxAVL t1
                      in armarAVL k v t1' t2
splitMaxAVL (N h k v E t2)  = (k, v, t2)
splitMaxAVL (N h k v t1 t2) =
            let (k', v', t1') = splitMaxAVL t1
             in (k', v', armarAVL k v t1' t2)
      ⋮
```

Mappings (BSTs)

- Implementaciones con AVLs (cont.)

```
  ⋮
lookupM E                    k v = Nothing
lookupM (N h k' v' t1 t2) k v =
          if   k==k'    then Just v
          else if k<k' then lookupM t1 k v
          else              lookupM t2 k v
  ⋮
```

- El código de lookupM es idéntico, pues no modifica el árbol

Mappings (AVLs)

- Implementaciones con AVLs (auxiliares)

```
      ⋮
-- ¡¡¡¡auxiliares para balanceo!!!! (NUEVAS desde BST)
armarAVL k v t1 t2    =
            if abs (altura t1 – altura t2) <= 1
             then  sJoinAVL k v t1 t2          -- Solo armar
             else  if (altura t1 == altura t2 + 2)
                      then rJoinAVL k v t1 t2  -- Rotar a der.
                      else lJoinAVL k v t1 t2  -- Rotar a izq.
      ⋮
```

Mappings (AVLs)

- Implementaciones con AVLs (auxiliares)

```
      ⋮
sJoinAVL k v t1 t2  =
            N (calcularAltura t1 t2) k v t1 t2
calcularAltura t1 t2 =
            1 + max (altura t1) (altura t2)
altura E = 0
altura (N h k v t1 t2) = h
      ⋮
```

Mappings (AVLs)

- Implementaciones con AVLs (auxiliares)

```
      ⋮
rJoinAVL k v (N hi ki vi tii tid) td   =
            if altura tii >= altura tid
             then  rSimpleR k v ki vi tii tid td
             else  rDobleR k v ki vi tii tid td
      ⋮
```

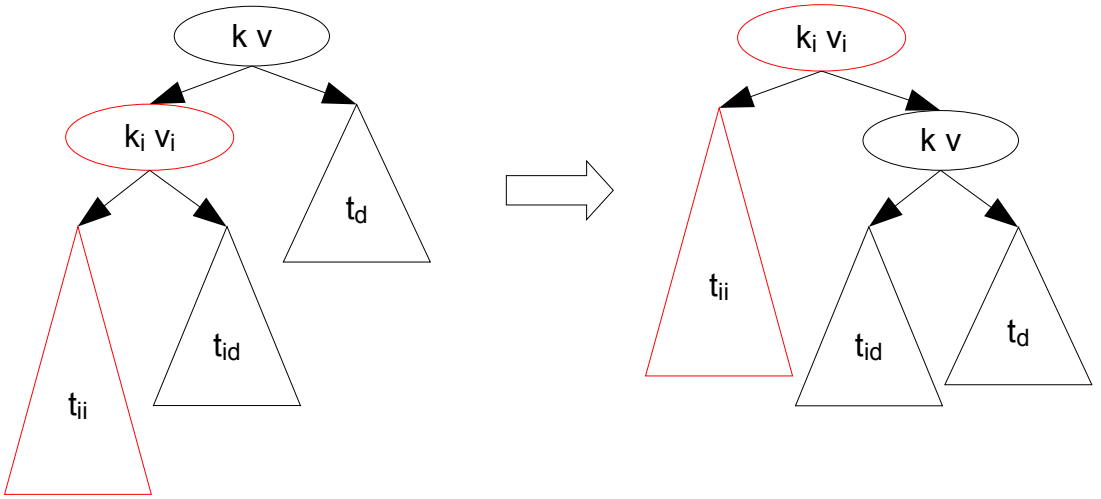
  - Si tii es más grande que tid, se hace rotación simple; sino, rotación doble (ver funciones a cont.)

Mappings (AVLs)

- Implementaciones con AVLs (auxiliares)

⋮

```
rSimpleR k v ki vi tii tid td  =
        sJoinAVL ki vi tii (sJoinAVL k v tid td)
```

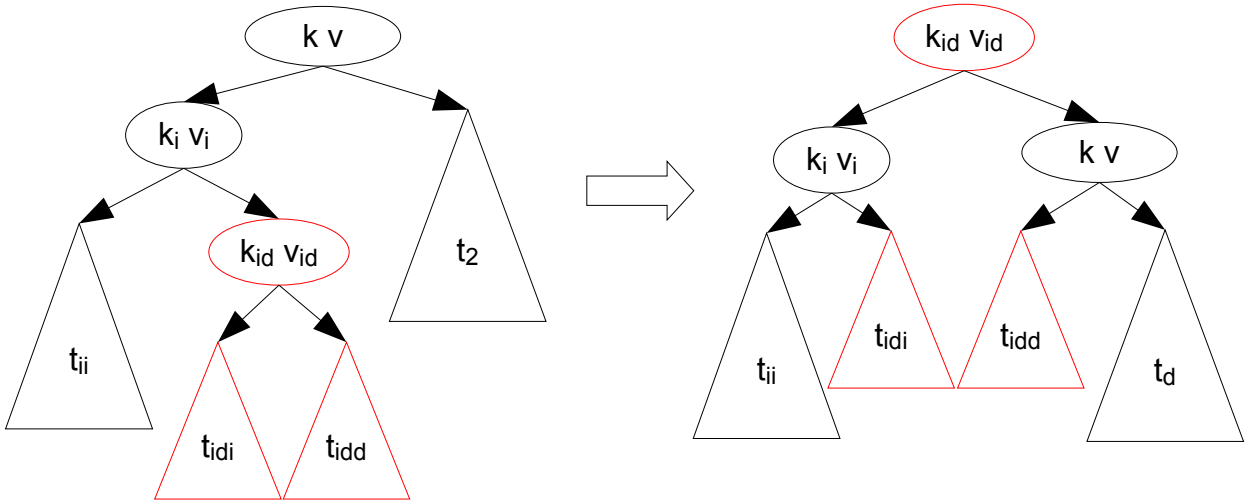


⋮

Mappings (AVLs)

- Implementaciones con AVLs (auxiliares)

```
  ⋮
rDobleR k v ki vi tii (N _ kid vid tidi tidd) td =
        sJoinAVL kid vid (sJoinAVL ki vi tii tidi)
                         (sJoinAVL k v tidd td)
  ⋮
```



Mappings (AVLs)

- Implementaciones con AVLs (cont.)
  - Las rotaciones a izquierda son simétricas
  - El código de balanceo es largo, pero no demasiado complicado
  - Los puntos de modificación respecto de BSTs son mínimos
  - El borrado NO es más difícil que en BSTs

Algoritmos sobre árboles

- Se puede destacar
  - el poder de la programación funcional para transmitir ideas sobre estructuras de datos
  - las propiedades estructurales de estructuras aparentemente complejas
  - el rol de la eficiencia en la complejidad de las implementaciones imperativas