1. Búsqueda en Árboles (Ejemplo Dungeon)
Este es el ejemplo introductorio para analizar la complejidad en árboles.
Haskell
```haskell
data Dir = Izq | Der

data Objeto = Armadura | Escudo | Maza -- etc.

data Dungeon = Armario
        | Habitacion Objeto Dungeon Dungeon

hayOroEn :: Dungeon -> [Dir] -> Bool
hayOroEn Armario _ = False
hayOroEn (Habitacion obj _ _) [] = esOro obj -- esOro no está definido, es conceptual
hayOroEn (Habitacion _ d1 d2) (d:ds) =
  case d of
    Izq -> hayOroEn d1 ds
    Der -> hayOroEn d2 ds

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

---

2. Set Implementado con Binary Search Tree (BST)
Aquí está la implementación del TAD Set utilizando un Árbol Binario de Búsqueda (BST).
Haskell
```haskell
-- Definición del tipo de dato para el árbol
data Tree a = EmptyT | NodeT a (Tree a) (Tree a)

-- Definición del tipo Set usando el árbol
data Set a = S (Tree a)
-- INV.REP.: en S t, t debe ser un BST.

-- Creación de un Set vacío
emptys :: Set a
emptys = S EmptyT

-- Pertenencia de un elemento en el Set
belongs :: Ord a => a -> Set a -> Bool
belongs x (S t) = buscarBST x t

buscarBST :: Ord a => a -> Tree a -> Bool
buscarBST _ EmptyT = False
buscarBST x (NodeT y ti td)
  | x == y    = True
  | x < y     = buscarBST x ti
  | otherwise = buscarBST x td

-- Adición de un elemento al Set
adds :: Ord a => a -> Set a -> Set a
```

```
adds x (S t) = S (insertarBST x t)

insertarBST :: Ord a => a -> Tree a -> Tree a
insertarBST x EmptyT = NodeT x EmptyT EmptyT
insertarBST x (NodeT y ti td)
  | x == y    = NodeT y ti td -- Ya existe, no se hace nada
  | x < y     = NodeT y (insertarBST x ti) td
  | otherwise = NodeT y ti (insertarBST x td)

-- Borrado de un elemento del Set
removeS :: Ord a => a -> Set a -> Set a
removeS x (S t) = S (borrarBST x t)

borrarBST :: Ord a => a -> Tree a -> Tree a
borrarBST _ EmptyT = EmptyT
borrarBST x (NodeT y ti td)
  | x < y     = NodeT y (borrarBST x ti) td
  | x > y     = NodeT y ti (borrarBST x td)
  | otherwise = rearmarBST ti td

rearmarBST :: Ord a => Tree a -> Tree a -> Tree a
rearmarBST EmptyT td = td
rearmarBST ti td = NodeT (maxBST ti) (delMaxBST ti) td

-- Funciones auxiliares para el borrado
maxBST :: Ord a => Tree a -> a
-- PRECOND: El árbol no es vacío
maxBST (NodeT x _ EmptyT) = x
maxBST (NodeT _ _ td) = maxBST td

delMaxBST :: Ord a => Tree a -> Tree a
-- PRECOND: El árbol no es vacío
delMaxBST (NodeT _ ti EmptyT) = ti
delMaxBST (NodeT x ti td) = NodeT x ti (delMaxBST td)

-- Conversión de Set a lista
set2list :: Set a -> [a]
set2list (S t) = inorder t

inorder :: Tree a -> [a]
inorder EmptyT = []
inorder (NodeT x ti td) = inorder ti ++ [x] ++ inorder td
```

---

3. Set Implementado con Árbol AVL

Esta es la implementación del TAD Set garantizando el balanceo con un árbol AVL.

Haskell

```
-- Definición del tipo de dato AVL
-- Se almacena la altura en cada nodo para eficiencia O(1)
data AVL a = EmptyAVL | NodeAVL Int a (AVL a) (AVL a)
-- INV.REP.: en NodeAVL h x ti td
```

```haskell
-- * h es la altura del árbol
-- * |altura(ti) - altura(td)| <= 1
-- * ti y td son AVLs
-- * cumple el invariante de BST

-- Definición del tipo Set usando el AVL
data Set a = S (AVL a)

-- Altura de un árbol AVL
heightAVL :: AVL a -> Int
heightAVL EmptyAVL = 0
heightAVL (NodeAVL h _ _ _) = h

-- Constructor inteligente para nodos AVL sin rotar
symAVL :: a -> AVL a -> AVL a -> AVL a
symAVL x ti td = NodeAVL (1 + max (heightAVL ti) (heightAVL td)) x ti td

-- Rotaciones (el código completo excede el alcance de la materia,
-- pero se presenta la estructura)

-- Rotación a la izquierda
leftAVL :: Ord a => a -> AVL a -> AVL a -> AVL a
leftAVL x (NodeAVL _ xi tii tid) td =
  if heightAVL tii >= heightAVL tid
    then -- Rotación simple
        symAVL xi tii (symAVL x tid td)
    else -- Rotación doble
        let (NodeAVL _ xid tidi tidd) = tid
        in symAVL xid (symAVL xi tii tidi) (symAVL x tidd td)

-- Rotación a la derecha
rightAVL :: Ord a => a -> AVL a -> AVL a -> AVL a
rightAVL x ti (NodeAVL _ xd tdi tdd) =
  if heightAVL tdd >= heightAVL tdi
    then -- Rotación simple
        symAVL xd (symAVL x ti tdi) tdd
    else -- Rotación doble
        let (NodeAVL _ xdi tdii tdid) = tdi
        in symAVL xdi (symAVL x ti tdii) (symAVL xd tdid tdd)

-- Constructor principal que decide si rotar
armarAVL :: Ord a => a -> AVL a -> AVL a -> AVL a
armarAVL x ti td
  | abs (hi - hd) <= 1 = symAVL x ti td
  | hi > hd          = leftAVL x ti td
  | otherwise        = rightAVL x ti td
  where
    hi = heightAVL ti
    hd = heightAVL td
```

```
-- Inserción en un AVL
insertAVL :: Ord a => a -> AVL a -> AVL a
insertAVL x EmptyAVL = NodeAVL 1 x EmptyAVL EmptyAVL
insertAVL x (NodeAVL _ y ti td)
  | x == y    = NodeAVL (heightAVL (NodeAVL 0 y ti td)) y ti td
  | x < y     = armarAVL y (insertAVL x ti) td
  | otherwise = armarAVL y ti (insertAVL x td)

-- Borrado en un AVL (simplificado, rearmarAVL completo es complejo)
deleteAVL :: Ord a => a -> AVL a -> AVL a
deleteAVL _ EmptyAVL = EmptyAVL
deleteAVL x (NodeAVL _ y ti td)
  | x < y     = armarAVL y (deleteAVL x ti) td
  | x > y     = armarAVL y ti (deleteAVL x td)
  | otherwise = rearmarAVL ti td

-- Re-armado post-borrado
rearmarAVL :: Ord a => AVL a -> AVL a -> AVL a
rearmarAVL EmptyAVL td = td
rearmarAVL ti td =
    let (m, ti') = splitMaxAVL ti
    in armarAVL m ti' td

-- Funciones auxiliares para borrado en AVL
maxAVL :: AVL a -> a
maxAVL (NodeAVL _ x _ EmptyAVL) = x
maxAVL (NodeAVL _ _ _ td) = maxAVL td

delMaxAVL :: Ord a => AVL a -> AVL a
delMaxAVL (NodeAVL _ _ ti EmptyAVL) = ti
delMaxAVL (NodeAVL _ x ti td) = armarAVL x ti (delMaxAVL td)

splitMaxAVL :: Ord a => AVL a -> (a, AVL a)
splitMaxAVL t = (maxAVL t, delMaxAVL t)

-- Implementación del Set
emptysAVL :: Set a
emptysAVL = S EmptyAVL

addsAVL :: Ord a => a -> Set a -> Set a
addsAVL x (S t) = S (insertAVL x t)

removeSAVL :: Ord a => a -> Set a -> Set a
removeSAVL x (S t) = S (deleteAVL x t)
```

---

4. PriorityQueue Implementada con Heap Binario

Implementación del TAD PriorityQueue utilizando un Heap Binario sobre una estructura de árbol.

Haskell

```
data Dir = Izq | Der
```

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

data PriorityQueue a = PQ [Dir] (Tree a)
-- INV.REP.: en (PQ pos t)
-- * t es un heap
-- * t es un árbol lleno
-- * pos es el camino INVERTIDO a la posición de inserción

-- Creación de una PQ vacía
emptyPQ :: PriorityQueue a
emptyPQ = PQ [] EmptyT

-- Chequeo si la PQ está vacía
isEmptyPQ :: PriorityQueue a -> Bool
isEmptyPQ (PQ _ EmptyT) = True
isEmptyPQ _           = False

-- Obtener el mínimo
findMinPQ :: PriorityQueue a -> a
-- PRECOND: La PQ no está vacía
findMinPQ (PQ _ (NodeT x _ _)) = x

-- Inserción en la PQ
insertPQ :: Ord a => a -> PriorityQueue a -> PriorityQueue a
insertPQ x (PQ pos t) = PQ (nextPos pos) (insertIn pos x t)

insertIn :: Ord a => [Dir] -> a -> Tree a -> Tree a
insertIn [] x _ = NodeT x EmptyT EmptyT
insertIn (Izq:ps) x (NodeT m ti td) = flotarIzq m (insertIn ps x ti) td
insertIn (Der:ps) x (NodeT m ti td) = flotarDer m ti (insertIn ps x td)

flotarIzq :: Ord a => a -> Tree a -> Tree a -> Tree a
flotarIzq m (NodeT m' ti' td') td =
  if m <= m'
    then NodeT m (NodeT m' ti' td') td
    else NodeT m' (NodeT m ti' td') td

flotarDer :: Ord a => a -> Tree a -> Tree a -> Tree a
flotarDer m ti (NodeT m' ti' td') =
  if m <= m'
    then NodeT m ti (NodeT m' ti' td')
    else NodeT m' ti (NodeT m ti' td')

nextPos :: [Dir] -> [Dir]
nextPos [] = [Izq]
nextPos (Izq:ps) = Der : ps
nextPos (Der:ps) = Izq : nextPos ps

-- Borrado del mínimo en la PQ
deleteMinPQ :: Ord a => PriorityQueue a -> PriorityQueue a
```

```
deleteMinPQ (PQ pos t) =
 let preP = prevPos pos
    (m', t') = splitAt preP t
 in PQ preP (hundir m' t')

splitAt :: [Dir] -> Tree a -> (a, Tree a)
splitAt [] (NodeT m _ _) = (m, EmptyT)
splitAt (Izq:ps) (NodeT m ti td) =
 let (m', ti') = splitAt ps ti
 in (m', NodeT m ti' td)
splitAt (Der:ps) (NodeT m ti td) =
 let (m', td') = splitAt ps td
 in (m', NodeT m ti td')

hundir :: Ord a => a -> Tree a -> Tree a
hundir m EmptyT = NodeT m EmptyT EmptyT
hundir m (NodeT mi ti' td') =
 case td' of
  EmptyT -> if m <= mi then NodeT m (NodeT mi ti' td') EmptyT
             else NodeT mi (hundir m ti') EmptyT
  (NodeT md tdi tdd) ->
   if m <= mi && m <= md
    then NodeT m (NodeT mi ti' td') (NodeT md tdi tdd)
   else if mi <= md
    then NodeT mi (hundir m ti') (NodeT md tdi tdd)
    else NodeT md (NodeT mi ti' td') (hundir m (NodeT md tdi tdd)) -- Ajuste conceptual

prevPos :: [Dir] -> [Dir]
prevPos [Izq] = []
prevPos (Der:ps) = Izq : ps
prevPos (Izq:ps) = Der : prevPos ps
```