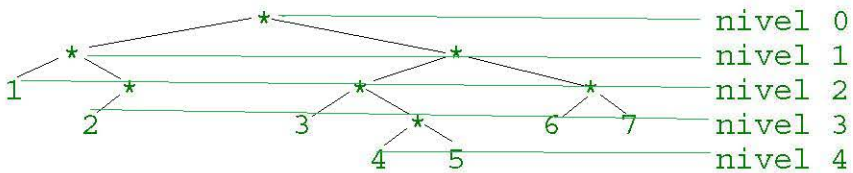
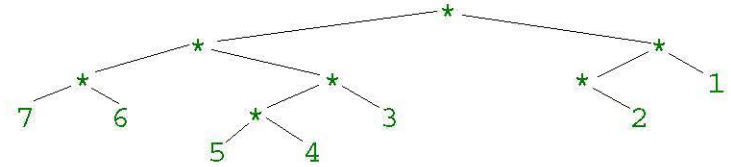
```haskell
--EJERCICIO 1
type Radio = Float
type Lado = Float
type Altura = Float
type Base = Float

data Circulo = Cir Radio
data Cuadrilatero = Cuad Lado | Rect Base Altura

class Figura a where
  perimetro :: a -> Float
  area :: a -> Float

-- Declarar los dos tipos Circulo y  Cuadrilatero
-- como instancias de la clase Figura, de modo que:
{-
*Main> area (Cir 3)
28.274334
*Main> perimetro (Cir 3)
18.849556
*Main> perimetro (Cuad 5)
20.0
*Main> area (Rect 5 8)
40.0
-}

-- EJERCICIO 2
-- Dadas las siguientes funciones auxiliares
quicksort :: Ord a => [a] -> [a]
quicksort [] = []
quicksort (x:xs) = quicksort (filter (<x) xs) ++
                   [x] ++
                   quicksort (filter (>=x) xs)
{-
*Main> quicksort [1,2,3,3,1,4,2,5,6,1,7,5,3]
[1,1,1,2,2,3,3,3,4,5,5,6,7]
-}
sinRepOrd :: (Eq a,Ord a) => [a] -> [a]
sinRepOrd = (foldr elim []) . quicksort
             where
             elim = \e rs -> e : case rs of
                                   [] -> []
                                   (x:xs) -> if x == e then xs else rs
{-
*Main> sinRepOrd [1,2,3,3,1,4,2,5,6,1,7,5,3]
[1,2,3,4,5,6,7]
-}
-- y el siguiente tipo algebraico
data Conj a = S [a]

--(2.1)
-- Declarar (Conj a) como instancia de la clase Eq de modo que:
{-
*Main> S [1,2,3,1,1,4,2,5] ==  S [5,2,3,1,4,4,2,5]
True
*Main> S [1,1,2] ==  S [1,2,3]
False
-}
```

```
--(2.2)
-- Declarar (Conj a) como instancia de la clase Show de modo que:
{-
*Main> S [5,2,3,1,4,4,2,5]
{1,2,3,4,5}
-}

--(2.3)
-- Definir las funciones que
-- calculen la unión y la intersección de dos conjuntos
        {-
        *Main> union (S [1,2,2]) (S [2,2,5])
        {1,2,5} -- el show definido arriba se aplica automáticamente
        *Main> intersec (S [1,2,2]) (S [2,2,5])
        {2} -- el show definido arriba se aplica automáticamente
        -}
-- calcule el cardinal de un conjunto,
        {-
        *Main> card (S [1,5,5,7,1,7])
        3
        -}
-- determine si un conjunto es subconjunto de otro.
        {-
        *Main> subConj (S [1,2,2]) (S [2,2,5])
        False
        *Main> subConj (S [5,2,2,3]) (S [2,5,5])
        False
        -}

-- EJERCICIO 3
-- Sea el siguiente tipo algebraico
data Arbol a =  Vacio
              | Hoja a
              | Arbol a :*: Arbol a
             deriving (Eq, Show)
--Ejemplos
arbol1 = (Hoja 1 :*: (Hoja 2 :*: Vacio))
         :*:
         ((Hoja 3 :*: ((Hoja 4 :*: Hoja 5)))
           :*:
          (Hoja 6 :*: Hoja 7))
{-
```




```
-}
arbol2 = ((Hoja 7 :*: Hoja 6)
           :*:
          (((Hoja 5 :*: Hoja 4)):*: Hoja 3))
         :*:
         ((Vacio :*: Hoja 2) :*: Hoja 1)
{-
```




```
-}
```

```haskell
-- (3.1)
-- Definir, usando ajuste de patrones, una función booleana que decida si
-- un arbol es la imagen en espejo de otro.
{-
*Main> espejo arbol1 arbol2
True
*Main> espejo (Hoja 4 :*: Hoja 5) (Hoja 5 :*: (Vacio :*: Hoja 4))
False
-}

-- (3.2)
-- Definir, usando ajuste de patrones, una función dados un número natural n
-- y un arbol, obtenga la lista de hojas que  tiene el arbol a nivel n.
-- Por ejemplo, para el arbol1 de arriba
-- para n = 0 ó n = 1, el resultado es []
-- para n = 2, el resultado es [1]
-- para n = 3, el resultado es [2,3,6,7]
-- para n = 4, el resulado es [4,5]
-- para n > 4, el resultado es []

-- (3.3)
-- Definir una función
-- plegar :: (b -> a) -> (a -> a -> a) -> a -> [b] -> a
-- tal que sirva para:
-- *Main> plegar id (+) 0 [1,2,3,4]
-- 10
-- y también para:
-- *Main> plegar Hoja (:*:) Vacio []
-- Vacio
-- *Main> plegar Hoja (:*:) Vacio [1]
-- Hoja 1
-- *Main> plegar Hoja (:*:) Vacio [1,2,3,4]
-- (Hoja 1 :*: Hoja 2) :*: (Hoja 3 :*: Hoja 4)

-- (3.4)
-- Utilizar la función plegar de (b) para definir una función miConcat que
-- haga lo mismo que la función predefinida concat. Es decir que por ejemplo:
-- *Main> miConcat [[1,2],[3,4,5],[6],[7,8,9]]
-- [1,2,3,4,5,6,7,8,9]

-- EJERCICIO 4
-- Dados los siguientes tipos de datos:
type Punto = (Float, Float)
{-
*Main> (0,0) < (2,0)
True
-}
type Arco = (Punto,Punto)
{-
*Main> ((2,0),(0,0)) > ((0,0),(2,0)) -- pero representan el mismo arco
True                                 -- porque son arcos no-dirigidos
-}
data Grafo = Vac | G [Arco] -- son grafos no-dirigido
     deriving (Eq,Ord,Show)
     --deriving (Ord,Show)  -- más tarde deberás cambiar el deriving

-- Ejemplos
g1 = G [((0,0),(1,1)),((2,0),(0,0)),((1,1),(3,1)),
        ((1,4),(2,4)),((1,4),(3,1))]
g2 = G [((3,1),(1,1)),((0,0),(1,1)),((0,0),(2,0)),
        ((1,4),(3,1)),((1,4),(2,4))]
g3 = G [((9,2),(3,6)),((3,6),(14,4)),((9,2),(23,13)),
```

```
        ((14,4),(5,22)),((5,22),(23,13))]
g4 = G [((3,6),(14,4)),((9,2),(3,6)),((23,13),(9,2)),
        ((5,22),(14,4)),((23,13),(5,22))]
g5 = G [((3,6),(14,4)),((9,2),(3,6)),((23,13),(9,2)),
        ((5,22),(14,4))]

-- La igualdad y el orden por "deriving" hacen que:
{-
*Main> Vac == G []
False
*Main> g1 <= g2
True
*Main> g2 <= g1
False
*Main> g1 == g2
False
*Main> g3 <= g4
False
*Main> g4 <= g3
True
*Main> g3 == g4
False
-}
-- Definir Grafo como instancia de Eq de modo que Vac y G [] sean iguales
-- y que (G xs == G ys) si y solo si xs e ys contienen los mismos arcos
-- con independencia del orden en xs e ys y de la orientación de cada arco
-- De modo que al cambiar el "deriving Eq" por tu instancia, ocurra que:
{-
*Main> g1 == g2
True
*Main> g1 == g3
False
*Main> g1 == g5
False
*Main> g3 == g4
True
-}
-- Indicación: tienes disponible el quicksort del EJERCICIO 2.
```