# CLASE IN HASKELL

```
length_ [] = 0
length_ (_:xs) = 1 + length_ xs
length :: [a] -> Int
map :: (a -> b) -> [a] -> [b]

elem _ []       = False
elem x (y:ys)   = x == y || elem x ys
elem :: Eq a => a -> [a] -> Bool

Clase
class  Eq a  where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool

data Person = Person {name :: String, cnp :: Integer}
instance Eq Person where
    Person name1 cnp1 == Person name2 cnp2 = name1 == name2 && cnp1
== cnp2
    p1 /= p2 = not (p1 == p2)
data BST a = Empty | Node a (BST a) (BST a)
instance Eq a => Eq (BST a) where
    Empty == Empty = True
    Node info1 l1 r1 == Node info2 l2 r2 = info1 == info2 && l1 ==
l2 && r1 == r2
    _ == _ = False
    t1 /= t2 = not (t1 == t2)

class  (Eq a) => Ord a  where
    compare              :: a -> a -> Ordering
    (<), (<=), (>), (>=) :: a -> a -> Bool
    max, min             :: a -> a -> a

    compare x y = if x == y then EQ
                  else if x <= y then LT
                  else GT

    x <  y = case compare x y of { LT -> True;  _ -> False }
    x <= y = case compare x y of { GT -> False; _ -> True }
    x >  y = case compare x y of { GT -> True;  _ -> False }
    x >= y = case compare x y of { LT -> False; _ -> True }

    max x y = if x <= y then y else x
    min x y = if x <= y then x else y

class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    a /= b = not (a == b)
```

```
instance Eq Person where
    Person name1 cnp1 == Person name2 cnp2 =
        name1 == name2 && cnp1 == cnp2

class  Eq a  where
    (==), (/=)           :: a -> a -> Bool
    x /= y               = not (x == y)
    x == y               = not (x /= y)
```

Clase predefinite
Ord – pentru tipuri care pot fi ordonate - definește funcții precum <, >, <=, etc.
Show – pentru tipuri care pot fi reprezentate ca String-uri - principala funcție este show. Această funcție este folosită și de consola GHCi atunci când afișează rezultatele.

Deriving

```
data BST a = Empty | Node a (BST a) (BST a) deriving (Eq)

treeMap :: (a -> b) -> BST a -> BST b
treeMap f Empty = Empty
treeMap f (BST info l r) = BST (f info) (treeMap f l) (treeMap f r)

data Maybe a = Nothing | Just a
maybeMap :: (a -> b) -> Maybe a -> Maybe b
maybeMap f Nothing = Nothing
maybeMap f (Just x) = Just (f x)

class Functor container where
    fmap :: (a -> b) -> container a -> container b

instance Functor BST where
   fmap f Empty = Empty
   fmap f (Node a l r) = Node (f a) (fmap f l) (fmap f r)


{-
    Clasa Container reprezintă o clasă folosită pentru enumerarea
    elementelor unei structuri de date (listă, arbore, graf, etc.)
-}

class Container t where
    contents :: t a -> [a]

{-
    Clasa Invertible reprezintă o clasă folosită pentru inversarea
    ordinii de apariție a elementelor unei structuri de date (listă,
arbore, etc.)
-}

class Invertible a where
    invert :: a -> a
```

```
{-
    BST (Binary Search Tree) reprezintă un tip de date ce modelează
    un arbore binar de căutare. Ați definit funcționalitățile
acestui
    tip de date în cadrul laboratorului 8.

    Funcționalitățile sunt deja definite în cadrul acestui
laborator.
-}

data BST a = BSTNod
    { vl :: a
    , lt :: BST a
    , rt :: BST a
    } | BSTNil

insertElem :: (Ord a, Eq a) => BST a -> a -> BST a
insertElem BSTNil elem = BSTNod elem BSTNil BSTNil
insertElem root@(BSTNod value left right) elem
  | value == elem = root
  | value < elem = BSTNod value left (insertElem right elem)
  | value > elem = BSTNod value (insertElem left elem) right

findElem :: (Ord a, Eq a) => BST a -> a -> Maybe a
findElem BSTNil _ = Nothing
findElem (BSTNod value left right) elem
  | value == elem = Just value
  | value < elem = findElem right elem
  | value > elem = findElem left elem

size :: BST a -> Int
size BSTNil = 0
size (BSTNod _ left right) = 1 + size left + size right

height :: BST a -> Int
height BSTNil = 0
height (BSTNod elem left right) = 1 + max (height left) (height
right)

inorder :: BST a -> [a]
inorder BSTNil = []
inorder (BSTNod elem left right) = inorder left ++ [elem] ++ inorder
right

{-
    Arbore folosit pentru testare
-}

root = foldl insertElem BSTNil [7, 4, 12, 2, 3, 1, 10, 15, 8]

{-
    1. Instanțiați Eq pentru tipul de date BST, prin care se
```

```
    verifică dacă doi arbori de acoperire sunt identici.
-}

instance Eq a => Eq (BST a) where
    (==) BSTNil BSTNil = True
    (==) (BSTNod e1 l1 r1) (BSTNod e2 l2 r2) = e1 == e2 && l1 == l2
&& r1 == r2
    (==) _ _ = False


{-
    2. Instanțiați Show pentru tipul de date BST.
    Fiecare nivel de adâncime în arbore va fi reprezentat de un număr
    corespunzător de tab-uri. De exemplu pentru nivelul 2 de adâncime
    se vor adăuga două tab-uri.

    Fiecare element din arbore va avea linia sa, adică câte un
element
    din arbore pe o linie.

-}

printLevel :: Show a => Char -> Int -> BST a -> [Char]
printLevel _ _ BSTNil = ""
printLevel tab level (BSTNod root left right) = replicate level tab
++ show root ++ "\n"
    ++ printLevel tab (level + 1) left
    ++ printLevel tab (level + 1) right

instance Show a => Show (BST a) where
    show BSTNil = ""
    show node@BSTNod{} = printLevel '\t' 0 node


{-
    3. Instanțiați Ord pentru tipul de date BST.
    Criteriul de comparare a doi arbori va fi după înălțimea lor
(funcția height).

    Trebuie implementată funcția (<=)
-}

instance Ord a => Ord (BST a) where
    (<=) t1 t2 = height t1 <= height t2

{-
    4. Instanțiați Invertible pentru tipul de date BST.
    Funcția invert, în acest caz, va inversa ordinea subarborilor.
-}

instance Invertible (BST a) where
    invert BSTNil = BSTNil
```

```
    invert (BSTNod a left right) = BSTNod a (invert right) (invert
left)


{-
    5. Instanțiați Functor pentru tipul de date BST.
    Funcția fmap este similară funcției map, prin care se aplică
    o funcție f tuturor elementelor din structură.
-}

instance Functor BST where
    fmap f BSTNil = BSTNil
    fmap f (BSTNod a left right) = BSTNod (f a) (fmap f left) (fmap
f right)


{-
    6. Instanțiați Foldable pentru tipul de date BST.
    Funcția foldr are aceeași funcționalitate atunci când
    ea este aplicată pe liste.

    Nu trebuie să implementați și foldl, clasa Foldable
    nu acoperă și această funcție.
-}

instance Foldable BST where
    foldr f acc BSTNil = acc
    foldr f acc (BSTNod value left right) = foldr f (f value newAcc)
left
        where newAcc = foldr f acc right


{-
    7. Instanțiați Container pentru tipul de date BST.
    Pentru implementarea funcției contents o să folosiți
    funcția foldr, implementată la exercițiul anterior.
-}

instance Container BST where
    contents tree = foldr (:) [] tree



{-
    8. Implementați sizeFold, care calculează numărul de elemente
    din cadrul unui arbore binar de căutare (există deja funcția
    size, care face același lucru, definită mai sus).
-}

sizeFold :: BST a -> Int
sizeFold tree = foldr (\_ acc -> acc + 1) 0 tree
```