Pràctica kd2nTrees Haskell - tardor 2015

Llenguatges de Programació (Universitat Politècnica de Catalunya)

```
-- Exemple de la prÃ ctica en format Kd2nTree
exampleSet :: Kd2nTree Point3d
exampleSet = Node ((Point3d 3.0 (-1.0) 2.1),[1,3]) [(Node ((Point3d 3.0 5.1 0.0),[2])
  [(Node ((Point3d 1.8 1.1 (-2.0)),[1,2]) [Empty,Empty,Empty,Empty]), (Node ((Point3d
  1.5 8.0 1.5),[1]) [Empty,Empty])]), (Node ((Point3d 3.0 (-1.7) 3.1),[1,2,3]) [Empty,
  Empty,Empty,Empty,Empty,Empty,Empty]), (Node ((Point3d 3.5 0.0 2.1),[3])
  [Empty,Empty]), (Node ((Point3d 3.5 2.8 3.1),[1,2]) [(Node ((Point3d 3.3 2.8 2.5),
  [3]) [Empty,Empty]), (Node ((Point3d 3.1 3.8 4.8),[1,3]) [Empty,Empty,Empty,Empty]),
  Empty, (Node ((Point3d 4.0 5.1 3.8),[2]) [Empty,Empty])])]

-- Exemple de la prÃ tica en format [([Double],[Int])]
exampleList :: [([Double],[Int])]
exampleList = [([3.0, -1.0, 2.1], [1, 3]), ([3.5, 2.8, 3.1], [1, 2]), ([3.5, 0.0, 2.1],
  [3]), ([3.0, -1.7, 3.1], [1, 2, 3]),([3.0, 5.1, 0.0], [2]), ([1.5, 8.0, 1.5], [1]),
  ([3.3, 2.8, 2.5], [3]), ([4.0, 5.1, 3.8], [2]),([3.1, 3.8, 4.8], [1, 3]), ([1.8, 1.1,
    -2.0], [1, 2])]


class (Eq p, Show p, Ord p) => Point p where
        sel :: Int -> p -> Double
        dim ::  p -> Int
        child :: p -> p -> [Int] -> Int
        dist :: p -> p -> Double
        list2Point :: [Double] -> p
        ptrans :: [Double] -> p -> p
        pscale :: Double -> p -> p

data Point3d = Point3d Double Double Double deriving (Eq, Show, Ord)

instance Point Point3d where
        sel 1 (Point3d x _ _) = x
        sel 2 (Point3d _ y _) = y
        sel 3 (Point3d _ _ z) = z

        dim _ = 3

        child e1 e2 cd = bin2dec (reverse (child_aux e1 e2 cd)) 0

        dist e1 e2 = sqrt ((((x2-x1)^2) + ((y2-y1)^2)) + ((z2-z1)^2))
                where
                        x1 = sel 1 e1
                        y1 = sel 2 e1
                        z1 = sel 3 e1
                        x2 = sel 1 e2
                        y2 = sel 2 e2
                        z2 = sel 3 e2

        list2Point (x:y:z:[]) = Point3d x y z

        ptrans (tx:ty:tz:[]) (Point3d x y z) = Point3d (x+tx) (y+ty) (z+tz)

        pscale scl (Point3d x y z) = Point3d (scl*x) (scl*y) (scl*z)

bin2dec :: [Int] -> Int -> Int
bin2dec [] _ = 0
bin2dec (x:xs) i
        | x == 1        = (2^i)+(bin2dec xs (i+1))
        | otherwise     = bin2dec xs (i+1)

child_aux :: (Point p) => p -> p -> [Int] -> [Int]
child_aux _ _ [] = []
child_aux e1 e2 (x:xs)
        | (sel x e1) <= (sel x e2)      = 0:child_aux e1 e2 xs
        | otherwise                                     = 1:child_aux e1 e2 xs
```

```
data Kd2nTree a = Node (a, [Int]) [Kd2nTree a] | Empty

instance (Eq a, Point a) => Eq (Kd2nTree a) where
        (==)    = equal

instance (Show a, Point a) => Show (Kd2nTree a) where
        show    = printTree



equal :: Eq a => Kd2nTree a -> Kd2nTree a -> Bool
equal Empty Empty = True
equal (Node (n1, cd1) fills1) Empty = False
equal Empty (Node (n2, cd2) fills2) = False
equal (Node (n1, cd1) fills1) (Node (n2, cd2) fills2)
        | (n1 == n2) && (cd1 == cd2)    = equalList fills1 fills2
        | otherwise                                     = False

equalList :: Eq a => [Kd2nTree a] -> [Kd2nTree a] -> Bool
equalList [] [] = True
equalList t1@(x:xs) t2@(y:ys)
        | (length t1) /= (length t2)    = False
        | otherwise                                     = (equal x y) && (equalList xs ys)



printTree :: Point a => Kd2nTree a -> String
printTree t = printTree_aux t 0

printTree_aux :: Point a => Kd2nTree a -> Int -> String
printTree_aux Empty _ = ""
printTree_aux (Node (n, cd) fills) tb = (printPoint n 1)++" "++(show cd)++(printTreeList 0 fills (tb))

printNTabs :: Int -> String
printNTabs 0 = ""
printNTabs n = "\t"++(printNTabs (n-1))

printTreeList :: Point a => Int -> [Kd2nTree a] -> Int -> String
printTreeList _ [] _ = ""
printTreeList n tl@(t:ts) tb
        | t == Empty    = printTreeList (n+1) ts tb
        | otherwise             = "\n"++(printNTabs tb)++"<"++(show n)++"> "++(printTree_aux t (tb+1))++(printTreeList (n+1) ts tb)

printPoint :: (Point a) => a -> Int -> String
printPoint p n
        | n == 1                = "("++(show (sel n p))++(printPoint p (n+1))
        | n < (dim p)   = ","++(show (sel n p))++(printPoint p (n+1))
        | n == (dim p)  = ","++(show (sel n p))++")"



insert :: Point a => Kd2nTree a -> a -> [Int] -> Kd2nTree a
insert Empty p cd = Node (p, cd) (makeNEmpties (2^(length cd)))
insert (Node (p0, cd0) fills) p cd
        | pos == Empty  = (Node (p0, cd0) (insertInPosition fills n (Node (p, cd) emptyChildren)))
        | otherwise     = (Node (p0, cd0) (insertInPosition fills n (insert pos p cd)))
        where
                n = child p p0 cd0
                pos = fills!!n
                emptyChildren = makeNEmpties (2^(length cd))

insertInPosition :: Point p => [Kd2nTree p] -> Int -> Kd2nTree p -> [Kd2nTree p]
insertInPosition l n p = (take n l)++[p]++(drop (n+1) l)

makeNEmpties :: Point p => Int -> [Kd2nTree p]
```

```
makeNEmpties 0 = []
makeNEmpties n = Empty:makeNEmpties (n-1)



build :: Point p => [(p, [Int])] -> Kd2nTree p
build [] = Empty
build tl = buildChildren (insert Empty p0 cd0) (tail tl)
        where (p0, cd0) = head tl

buildChildren :: Point p => Kd2nTree p -> [(p, [Int])] -> Kd2nTree p
buildChildren tree [] = tree
buildChildren root tl = buildChildren (insert root p cd) (tail tl)
        where (p, cd) = head tl

buildIni :: Point p => [([Double], [Int])] -> Kd2nTree p
buildIni tl = buildChildrenIni (insert Empty (list2Point p0) cd0) (tail tl)
        where (p0, cd0) = head tl

buildChildrenIni :: Point p => Kd2nTree p -> [([Double], [Int])] -> Kd2nTree p
buildChildrenIni tree [] = tree
buildChildrenIni root tl = buildChildrenIni (insert root ((list2Point p)) cd) (tail tl)
        where (p, cd) = head tl



get_all :: Point p => Kd2nTree p -> [(p, [Int])]
get_all Empty = []
get_all (Node (p, cd) fills) = [(p, cd)]++(get_allChildren fills)

get_allChildren :: Point p => [Kd2nTree p] -> [(p, [Int])]
get_allChildren [] = []
get_allChildren (t:ts) = (get_all t)++(get_allChildren ts)



remove :: Point p => Kd2nTree p -> p -> Kd2nTree p
remove Empty _ = Empty
remove t@(Node (pt, cd) fills) p
        | p == pt       = build (get_allChildren fills)
        | otherwise = buildChildren newFather (get_allChildren (removeChildren fills p))
        where newFather = Node (pt, cd) (makeNEmpties (2^(length cd)))

removeChildren :: Point p => [Kd2nTree p] -> p -> [Kd2nTree p]
removeChildren [] _ = []
removeChildren (t:ts) p = (remove t p):(removeChildren ts p)



contains :: Point p => Kd2nTree p -> p -> Bool
contains Empty _ = False
contains (Node (pt, cd) fills) p
        | pt == p       = True
        | otherwise = containsChildren fills p

containsChildren :: Point p => [Kd2nTree p] -> p -> Bool
containsChildren [] _ = False
containsChildren (t:ts) p = (contains t p) || containsChildren ts p



nearest :: Point p => Kd2nTree p -> p -> p
nearest tree p = points!!n
        where
                points = map (fst) (get_all tree)
                distances = map (dist p) points
```

```
                n = indexOf (minimum distances) distances 0
                indexOf i l n
                        | i == (head l)         = n
                        | otherwise             = indexOf i (tail l) (n+1)


allinInterval :: Point p => Kd2nTree p -> p -> p -> [p]
allinInterval Empty _ _ = []
allinInterval tree p1 p2 = filter (p1<=) (filter (p2>=) points)
        where points = sortlist (map (fst) (get_all tree))

sortlist :: Point p => [p] -> [p]
sortlist [] = []
sortlist l@(x:xs) = (sortlist a)++[x]++(sortlist b)
  where (a, b) = splitByP xs x

splitByP :: Point p => [p] -> p -> ([p],[p])
splitByP [] p = ([],[])
splitByP l@(x:xs) p
  | x >= p      = (a, x:b)
  | otherwise   = (x:a, b)
  where (a, b) = splitByP xs p



kdmap :: Point p => (p -> q) -> Kd2nTree p -> Kd2nTree q
kdmap _ Empty = Empty
kdmap f (Node (p, cd) fills) = Node (f p, cd) (kdmapChildren f fills)

kdmapChildren :: Point p => (p -> q) -> [Kd2nTree p] -> [Kd2nTree q]
kdmapChildren _ [] = []
kdmapChildren f (t:ts) = (kdmap f t):kdmapChildren f ts

translation :: Point p => [Double] -> Kd2nTree p -> Kd2nTree p
translation [] tree = tree
translation _ Empty = Empty
translation trans tree = kdmap (ptrans trans) tree

scale :: Point p => Double -> Kd2nTree p -> Kd2nTree p
scale _ Empty = Empty
scale scl tree = kdmap (pscale scl) tree
```