99 questions/Solutions/63

From HaskellWiki

< 99 questions | Solutions

Construct a complete binary tree

A complete binary tree with height H is defined as follows:

- The levels 1,2,3,...,H-1 contain the maximum number of nodes (i.e 2**(i-1) at the level i)
- In level H, which may contain less than the maximum possible number of nodes, all the nodes are "left-adjusted". This means that in a levelorder tree traversal all internal nodes come first, the leaves come second, and empty successors (the nil's which are not really nodes!) come last.

Particularly, complete binary trees are used as data structures (or addressing schemes) for heaps.

We can assign an address number to each node in a complete binary tree by enumerating the nodes in level-order, starting at the root with number 1. For every node X with address A the following property holds: The address of X's left and right successors are 2*A and 2*A+1, respectively, if they exist. This fact can be used to elegantly construct a complete binary tree structure.

Write a predicate complete_binary_tree/2.

```
import Data.List

data Tree a = Empty | Branch a (Tree a) (Tree a)
        deriving (Show, Eq)

filled :: Tree a -> [[Bool]]
filled Empty = repeat [False]
filled (Branch _ l r) = [True] : zipWith (++) (filled l) (filled r)

completeBinaryTree :: Int -> Tree Char
completeBinaryTree n = generate_tree 1
  where generate_tree x
          | x > n     = Empty
          | otherwise = Branch 'x' (generate_tree (2*x)  )
                                   (generate_tree (2*x+1))

isCompleteBinaryTree :: Tree a -> Bool
isCompleteBinaryTree Empty = True
isCompleteBinaryTree t = and $ last_proper : zipWith (==) lengths powers
  where levels      = takeWhile or $ filled t
        -- The upper levels of the tree should be filled.
        -- Every level has twice the number of nodes as the one above it,
```

```
        -- so [1,2,4,8,16,...]
        lengths     = map (length . filter id) $ init levels
        powers      = iterate (2*) 1
        -- The last level should contain a number of filled spots,
        -- and (maybe) some empty spots, but no filled spots after that!
        last_filled = map head $ group $ last levels
        last_proper = head last_filled && (length last_filled) < 3
```

The "generate_tree" method here creates Node number "x", and then creates x's children nodes. By the property described above, these nodes are numbers 2*x and 2*x+1. If x > n, then the node is only created as Empty, and does not create children nodes.

Alternative solution which constructs complete binary trees from a given list using local recursion (also includes a lookup function as per the Prolog solution):

```
completeBinaryTree :: Int -> a -> Tree a
completeBinaryTree n = cbtFromList . replicate n

cbtFromList :: [a] -> Tree a
cbtFromList xs = let (t, xss) = cbt (xs:xss) in t
  where cbt ((x:xs):xss) =
              let (l, xss') = cbt xss
                  (r, xss'') = cbt xss'
              in  (Branch x l r, xs:xss'')
        cbt _ = (Empty, [])

lookupIndex :: Tree a -> Integer -> a
lookupIndex t = lookup t . path
    where lookup Empty _ = error "index to large"
          lookup (Branch x _ _) [] = x
          lookup (Branch x l r) (p:ps) = lookup (if even p then l else r) ps

          path = reverse . takeWhile (>1) . iterate (`div` 2) . (1+)
```

Here is a similar simple solution for the predicate. In a complete balanced binary tree, the first empty node should come after the last populated node, in a left-to-right breadth-first search. This simply checks that.

```
isCompleteBinaryTree t = b > a where
  (a, b) = maxmin t 1
  maxmin Empty m = (0, m)
  maxmin (Branch _ l r) m = (max m $ max max_l max_r, min min_l min_r) where
    (max_l, min_l) = maxmin l (2*m)
    (max_r, min_r) = maxmin r (2*m + 1)
```

We can also implement
`isCompleteBinaryTree`
by generate the equal-sized CBT and compare.

```
treeNodes Empty = 0
treeNodes (Branch _ left right) = 1 + treeNodes left + treeNodes right
```

```
treeEqual Empty Empty = True
treeEqual (Branch _ l1 r1) (Branch _ l2 r2) =
    (treeEqual l1 l2) && (treeEqual r1 r2)
treeEqual _ _ = False

isCompleteBinaryTree t = treeEqual t $ completeBinaryTree $ treeNodes t
```

An alternative implementation for `isCompleteBinaryTree`:

```
completeHeight Empty = Just 0
completeHeight (Branch _ l r) = do
        hr <- completeHeight r
        hl <- completeHeight l
        if (hl == hr) || (hl - hr == 1)
                then return $ 1+hl
                else Nothing

isCompleteBinaryTree = (/=Nothing) . completeHeight
--This solution includes trees such as

       'x'
       / \
     'x' 'x'
     /   /
   'x'  'x'

--as complete binary trees but notice that address of left child is not necessarily twice tha
```

This uses a helper method `completeHeight` which calculates the height of a complete binary tree, or returns `Nothing` if it is not complete.
