99 questions/Solutions/55

From HaskellWiki

< 99 questions | Solutions

(**) Construct completely balanced binary trees

In a completely balanced binary tree, the following property holds for every node: The number of nodes in its left subtree and the number of nodes in its right subtree are almost equal, which means their difference is not greater than one.

Write a function cbal-tree to construct completely balanced binary trees for a given number of nodes. The predicate should generate all solutions via backtracking. Put the letter 'x' as information into all nodes of the tree.

```
cbalTree :: Int -> [Tree Char]
cbalTree 0 = [Empty]
cbalTree n = let (q, r) = (n - 1) `quotRem` 2
    in [Branch 'x' left right | i     <- [q .. q + r],
                                left  <- cbalTree i,
                                right <- cbalTree (n - i - 1)]
```

This solution uses a list comprehension to enumerate all the trees, in a style that is more natural than standard backtracking.

The base case is a tree of size 0, for which `Empty` is the only possibility. Trees of size `n == 1` or larger consist of a branch, having left and right subtrees with sizes that sum up to `n - 1`. This is accomplished by getting the quotient and remainder of `(n - 1)` divided by two; the remainder will be 0 if `n` is odd, and 1 if `n` is even. For `n == 4`, `(q, r) = (1, 1)`.

Inside the list comprehension, `i` varies from `q` to `q + r`. In our `n == 4` example, `i` will vary from 1 to 2. We recursively get all possible left subtrees of size `[1..2]`, and all right subtrees with the remaining elements.

When we recursively call `cbalTree 1`, `q` and `r` will both be 0, thus `i` will be 0, and the left subtree will simply be `Empty`. The same goes for the right subtree, since `n - i - 1` is 0. This gives back a branch with no children--a "leaf" node:

```
> cbalTree 1
[Branch 'x' Empty Empty]
```

The call to `cbalTree 2` sets `(q, r) = (0, 1)`, so we'll get back a list of two possible subtrees. One has an empty left branch, the other an empty right branch:

```
> cbalTree 2
[
  Branch 'x' Empty (Branch 'x' Empty Empty),
  Branch 'x' (Branch 'x' Empty Empty) Empty
]
```

In this way, balances trees of any size can be built recursively from smaller trees.

A slightly more efficient version of this solution, which never creates the same tree twice:

```
cbalTree 0 = [Empty]
cbalTree 1 = [leaf 'x']
cbalTree n = if n `mod` 2 == 1 then
             [ Branch 'x' l r | l <- cbalTree ((n - 1) `div` 2),
                                r <- cbalTree ((n - 1) `div` 2) ]
             else
             concat [ [Branch 'x' l r, Branch 'x' r l] | l <- cbalTree ((n - 1) `div` 2),
                                                         r <- cbalTree (n `div` 2) ]
```

Another approach is to create a list of all possible tree structures with a given number of nodes, and then filter that list on whether or not the tree is balanced.

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

main = putStrLn $ concatMap (\t -> show t ++ "\n") balTrees
    where balTrees = filter isBalancedTree (makeTrees 'x' 4)

isBalancedTree :: Tree a -> Bool
isBlanacedTree Empty = True
isBalancedTree (Branch _ l r) = abs (countBranches l - countBranches r) ≤ 1
                                && isBalancedTree l && isBalancedTree r
isBalancedTree _ = False

countBranches :: Tree a -> Int
countBranches Empty = 0
countBranches (Branch _ l r) = 1 + countBranches l + countBranches r

-- makes all possible trees filled with the given number of nodes
-- and fill them with the given value
makeTrees :: a -> Int -> [Tree a]
makeTrees _ 0 = []
makeTrees c 1 = [leaf c]
makeTrees c n = lonly ++ ronly ++ landr
    where lonly  = [Branch c t Empty | t <- smallerTree]
          ronly = [Branch c Empty t | t <- smallerTree]
          landr = concat [[Branch c l r | l <- fst lrtrees, r <- snd lrtrees] | lrtrees <- tr
          smallerTree = makeTrees c (n-1)
          treeMinusTwo = [(makeTrees c num, makeTrees c (n-1-num)) | num <- [0..n-2]]
```

While not nearly as neat as the previous solution, this solution uses some generic binary tree methods that could be useful in other contexts.