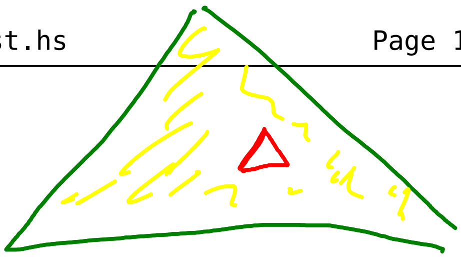


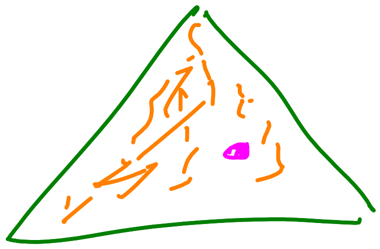
```
-- UNIT TEST ---------------------------S

{-

-- 1
-- finds triangle
```

in sea of zeros

```
> o = Node (Node (Node (Leaf 0) 0 (Leaf 0)) 0 (Node (Leaf 0) 0 (Leaf 0))) 0
(Node (Node (Node (Leaf 0) 0 (Leaf 0)) 0 (Node (Leaf 0) 0 (Leaf 0))) 0 (Node
(Node (Node (Leaf 0) 0 (Leaf 0)) 1 (Node (Leaf 0) 0 (Leaf 0))) 1 (Node (Node
(Leaf 0) 0 (Leaf 0)) 1 (Node (Leaf 0) 0 (Leaf 0)))))

> 3 == maxPath o

True
```

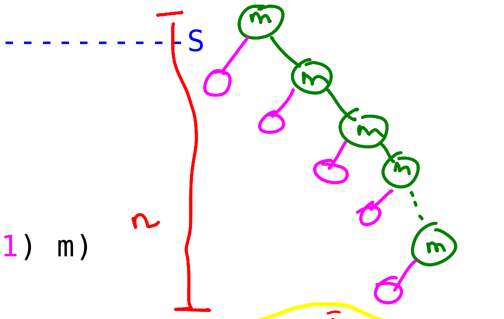
checks if my solution is not ignoring hidden triangles

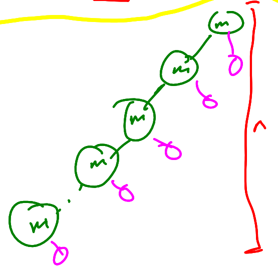
```
-- 2
-- finds zero in a sea of negatives

> z = Node (Node (Node (Leaf (-1)) (-1) (Leaf (-1))) (-1) (Node (Leaf (-1))
(-1) (Leaf (-1)))) (-1) (Node (Node (Node (Leaf (-1)) (-1) (Leaf (-1))) (-1)
(Node (Leaf (-1)) (-1) (Leaf (-1)))) (-1) (Node (Node (Node (Leaf (-1)) (-1)
(Leaf (-1))) (-1) (Node (Leaf (-1)) (-1) (Leaf (-1)))) 0 (Node (Node (Leaf
(-1)) (-1) (Leaf (-1))) (-1) (Node (Leaf (-1)) (-1) (Leaf (-1))))))

> 0 == maxPath z

True

-}

-- UNIT TEST ---------------------------E
```

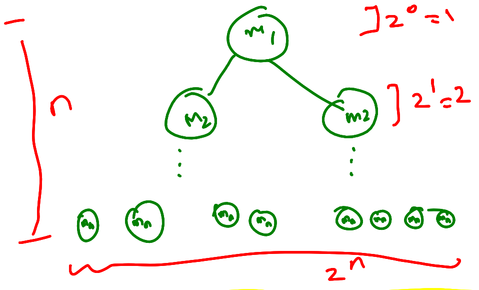


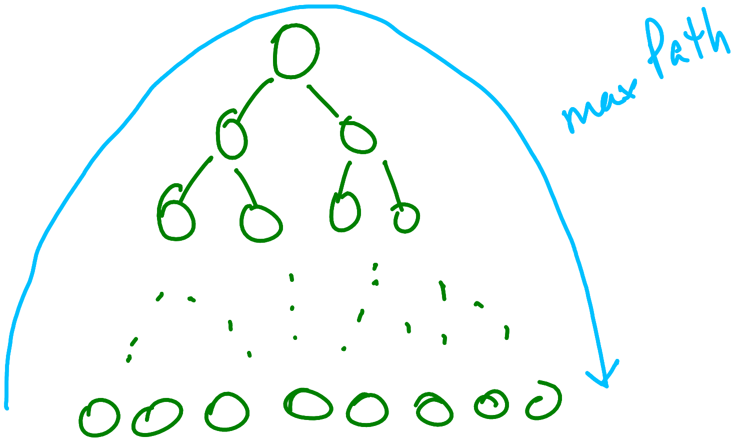
Testing for handling negatives correctly and identifying the 1 +ve value

```haskell
-- PROPERTY TEST ------------------------------------------S

-- 1
treeGenRight :: Int -> Int -> Tree Int
treeGenRight n m
  | n <= 1    = Leaf m
  | otherwise = Node (Leaf 0) m (treeGenRight (n - 1) m)


treeGenLeft :: Int -> Int -> Tree Int
treeGenLeft n m
  | n <= 1    = Leaf m
  | otherwise = Node (treeGenLeft (n - 1) m) m (Leaf 0)


treeGenLR :: Int -> Int -> Tree Int
treeGenLR n m
  | n <= 1    = Leaf m
  | otherwise = Node t m t
                where
                  t = treeGenLR (n - 1) m


qcBoundary :: Int -> Int -> Property
qcBoundary n m =
  m >= 0 && n >= 0 && n < 20 ==>
    (maxPath (treeGenLeft n m) + maxPath (treeGenRight n m) - m)
    ==
    maxPath (treeGenLR n m)
```



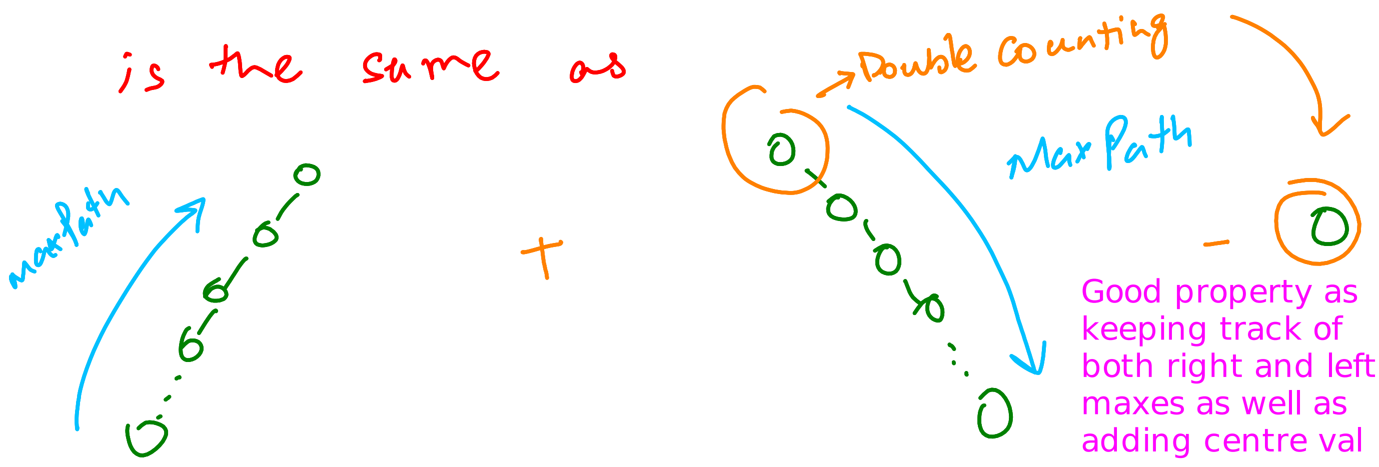






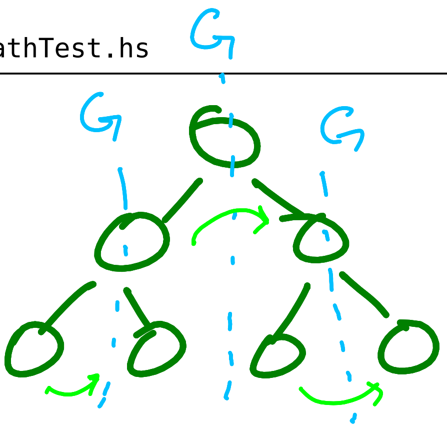
* may not work for other arbitrary tree generators

is the same as

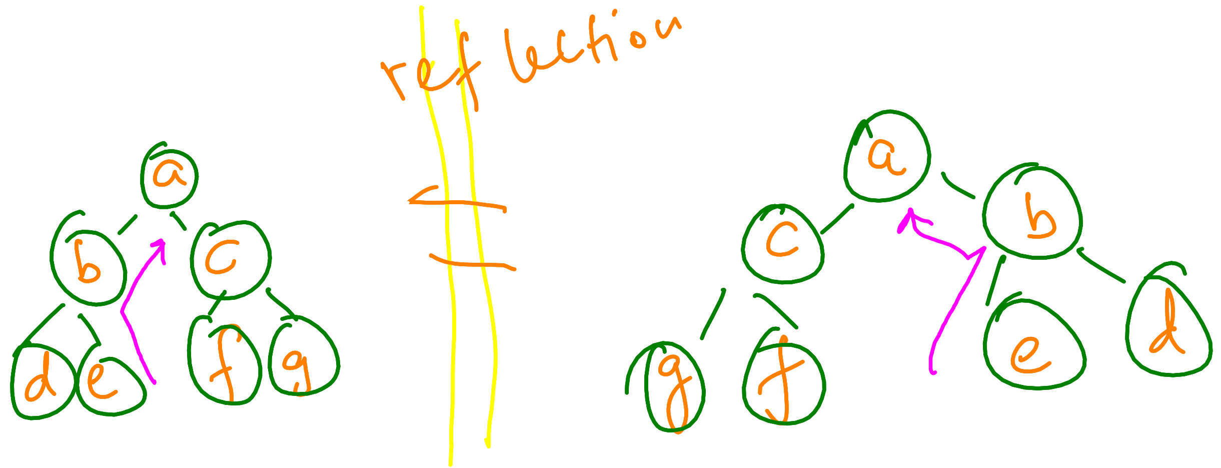


Good property as keeping track of both right and left maxes as well as adding centre val

```haskell
--2
treeGen :: [Int] -> Tree Int
treeGen [m] = Leaf m
treeGen (m:ms) = Node t m t
                 where
                   t = treeGen ms



mirrorTree :: Tree Int -> Tree Int
mirrorTree (Leaf a) = Leaf a
mirrorTree (Node l a r) = Node (mirrorTree r) a (mirrorTree l)



qcMirror :: [Int] -> Property
qcMirror xs =
  not (null xs) && length xs <= 15 ==>
  ml == ml'
  where
    l = treeGen xs
    ml = maxPath l
    l' = mirrorTree l
    ml' = maxPath l'

-- PROPERTY TEST ------------------------------------------E
```



& can use arbitrary tree generators



But max remains same as perfect mirror image

This is a good property to test our code if it is not being biased to one side