```
 -- setting the "warn-incomplete-patterns" flag asks GHC to warn you
-- about possible missing cases in pattern-matching definitions
{-# OPTIONS_GHC -fwarn-incomplete-patterns #-}

-- see https://wiki.haskell.org/Safe_Haskell
{-# LANGUAGE Safe #-}

module Assessed4 (choose , simulate , cut , shuffle , riffles , permute , genTree)
where

import Types

---------------------------------------------------------------------------------
---------------- DO **NOT** MAKE ANY CHANGES ABOVE THIS LINE --------------------
---------------------------------------------------------------------------------

-- import System.Random
-- import Control.Monad.Identity
-- import Control.Monad.State
{- Exercise 1 -}

choose :: PickingMonad m => [a] -> m a
choose list = do
                i <- pick 0 (length list - 1)
                pure (list !! i)

simulate :: Monad m => m Bool -> Integer -> m Integer
simulate mBool 0 = return 0
simulate mBool num = do
                        m <- mBool
                        if m == True then (fmap (+1) (simulate mBool (num - 1)))
                        else (simulate mBool (num - 1))


-- simulate :: Monad m => m Bool -> Integer -> m Integer
-- simulate mBool 0 = return 0
-- simulate mBool num = do
--                         m <- mBool
--                         simulated <- simulate mBool (num - 1)
--                         if m == True then return (simulated + 1)
--                         else return simulated


cut :: PickingMonad m => [a] -> m ([a],[a])
cut (x:xs) = do
                y <- pick 0 (length (x:xs))
                pure (splitAt y (x:xs))
cut [] = pure ([], [])


shuffle :: PickingMonad m => ([a],[a]) -> m [a]
shuffle ([], []) = return []
shuffle ([], sec) = return sec
shuffle (fir, []) = return fir
shuffle ((fir:firs) ,(sec:secs)) = do
                        choice <- pick 0 (length (fir:firs) + length (sec:secs) -
1)
                        if choice < (length (fir:firs)) then (fmap (fir:) (shuffle
(firs, (sec:secs))))
```

```
                        else (fmap (sec:) (shuffle ((fir:firs), (secs))))


-- shuffle :: PickingMonad m => ([a],[a]) -> m [a]
-- shuffle ([], []) = return []
-- shuffle ([], sec) = return sec
-- shuffle (fir, []) = return fir
-- shuffle ((fir:firs) ,(sec:secs)) = do
--                         choice <- pick 0 (length (fir:firs) + length (sec:secs)
- 1)
--                         if choice < (length (fir:firs)) then do
--                                                                 shuffled <-
shuffle (firs, (sec:secs))
--                                                                 return (fir:
shuffled)
--                         else do
--                                 shuffled <- shuffle ((fir:firs), (secs))
--                                 return (sec:shuffled)

-- shuffle :: PickingMonad m => ([a],[a]) -> m [a]
-- shuffle (fir, sec) = shuff2 (length fir) (length sec) (fir, sec)


-- shuff2 :: PickingMonad m => Int -> Int -> ([a],[a]) -> m [a]
-- shuff2 _ _ ([], []) = return []
-- shuff2 _ _ ([], sec) = return sec
-- shuff2 _ _ (fir, []) = return fir
-- shuff2 len1 len2 ((fir:firs) ,(sec:secs)) = do
--                         choice <- pick 0 (len1 + len2 - 1)
--                         if choice < (len1) then (fmap (fir:) (shuff2 (len1 - 1)
(len2) (firs, (sec:secs))))
--                         else (fmap (sec:) (shuff2 (len1) (len2 - 1) ((fir:firs),
(secs))))



riffles :: PickingMonad m => ([a] -> m ([a],[a])) -> (([a],[a]) -> m [a]) -> Int ->
[a] -> m [a]
riffles cf sf 0 xs = (>>=) (cf xs) sf
riffles cf sf n xs = do
                        bound <- (>>=) (cf xs) sf
                        riffles cf sf (n-1) (bound)

-- riffles :: PickingMonad m => ([a] -> m ([a],[a])) -> (([a],[a]) -> m [a]) -> Int
-> [a] -> m [a]
-- riffles cf sf 0 xs = sf (cf xs)
-- riffles cf sf n xs = do
--                         bound <- (>>=) (cf xs) sf
--                         riffles cf sf (n-1) (bound)


-- riffles :: PickingMonad m => ([a] -> m ([a],[a])) -> (([a],[a]) -> m [a]) -> Int
-> [a] -> m [a]
-- riffles cf sf 0 xs = do
--                         beenCut <- cf xs
--                         sf beenCut
-- riffles cf sf n xs = do
--                         beenCut <- cf xs
--                         beenShuffled <- sf beenCut
```

```
--                          riffles cf sf (n-1) (beenShuffled)

-- riffles :: PickingMonad m => ([a] -> m ([a],[a])) -> (([a],[a]) -> m [a]) -> Int
-> [a] -> m [a]
-- riffles cf sf 0 xs = do
--                          result <- (>>=) (cf xs) sf
--                          result
-- riffles cf sf n xs = riffles cf sf (n-1) ((>>=) (cf xs) sf)

permute :: PickingMonad m => [a] -> m [a]
-- permute = undefined
permute [] = return []
permute (x:xs) =
                do
                    i <- pick 0 (length xs)
                    fmap (((x:xs) !! i):) (permute (delete i (x:xs)))

delete :: Int -> [a] -> [a]
delete _ []     = []
delete i (x:xs) | i == 0 = xs
                | otherwise = x : delete (i-1) xs

-- genTree :: PickingMonad m => [a] -> m (Bin a)
-- genTree [list] = return (L list)
-- genTree (l:ls) = do
--                      list <- permute (l:ls)
--                      mkTree (list)

-- genTree = undefined

-- mkTree2 :: PickingMonad m => [a] -> m (Bin a)
-- mkTree2 [x] = return (L x)
-- mkTree2 (x:xs) = do
--                      i <- pick 0 (length xs)
--                      f <- pick 0 1
--                      tree <- mkTree2 xs
--                      if f == 0 then (fmap (\((L x), binn) -> (B (L x) (binn))) (L
x, tree))
--                      else (fmap (\((L x), binn) -> (B  (binn) (L x))) (L x,
tree))


-- mkTree :: PickingMonad m => [a] -> m (Bin a)
-- mkTree [x] = return (L x)
-- mkTree (x:xs) =
--                  do
--                      tree <- mkTree xs
--                      i <- pick 0 1
--                      if i == 0 then return (B (L  x) (tree))
--                      else return (B (tree) (L x))


-- mkTree :: PickingMonad m => Int -> Int -> Int -> [a] -> m [a]
-- mkTree _ _ _ [x] = return [x]
-- mkTree intNodes n l0 list | n == intNodes = return list
--                           | otherwise = do
--                                         cX <- pick 0 (4*n + 1)
--                                         let n2 = n + 1
--                                         let b = cX `mod` 2
```

```
--                                          let k = cX `div` 2
--                                          newList1 <- return (replace list (2 * n2
- b) (2 * n2))
--                                          newList2 <- return (replace list (2*n2 -
1 + b) (newList1 !! k))
--                                          newList3 <- return (replace list (k)
(2*n2 - 1))
--                                          mkTree intNodes n2 l0 newList3


replace :: [a] -> Int -> a -> [a]
replace [] _ _  = []
replace (x:xs) i val = if i > (length (x:xs)) - 1 then (x:xs)
                        else
                            do
                            let (x1, x2) = splitAt i (x:xs)
                            x1 ++ val : (tail x2)




-- replaceLeaf :: PickingMonad m => a -> Bin a -> m (Bin a)
-- replaceLeaf x tree = do
--                       i <- pick 0 1
--                       if i == 0 then return (B (L x) (tree))
--                       else return (B (tree) (L x))

-- genTree3 :: PickingMonad m => [a] -> m (Bin a)
-- genTree3 [x] = return (L x)
-- genTree3 (x:xs) = do
--                       ran <- genTree3 xs
--                       goToRandomLeaf x (ran)


--[1,2,3]
--ran <- genTree[2,3]
--      --ran <- genTree[3]
--             --L 3
--        ran <- L3
--        goToRandomLeaf 2 (L 3)
--ran <- (B (L 2) (L 3),1 % 2),(B (L 3) (L 2),1 % 2)
--goToRandomLeaf 1 ((B (L 2) (L 3),1 % 2),(B (L 3) (L 2),1 % 2))



-- goToRandomLeaf :: PickingMonad m => a -> Bin a -> m (Bin a)
-- goToRandomLeaf val (L x) = (genTree2 val (L x))
-- goToRandomLeaf val (B (left) (right)) = do
--                                              i <- pick 0 1
--                                              if i == 0 then do
--                                                              newLeft <-
goToRandomLeaf val left
--                                                              return (B (newLeft)
(right))
--                                              else do
--                                                      newRight <- goToRandomLeaf
val right
--                                                      return (B (left) (newRight))
```

```
-- goToRandomLeaf :: PickingMonad m => a -> Bin a -> m (Bin a)
-- goToRandomLeaf val (L x) = (replaceLeaf val (L x))
-- goToRandomLeaf val (B (left) (right)) = do
--                                               i <- pick 0 2
--                                               if i == 0 then do
--                                                               newLeft <-
goToRandomLeaf val left
--                                                               return (B (newLeft)
(right))
--                                               else if i == 1 then do
--                                                               newRight <-
goToRandomLeaf val right
--                                                               return (B (left)
(newRight))
--                                               else do
--                                                           j <- pick 0 1
--                                                           if j == 0 then return (B
(B (left) (right)) (L val) )
--                                                           else return (B  (L val)
(B (left) (right)))


genTree :: PickingMonad m => [a] -> m (Bin a)
genTree [x] = return (L x)
genTree (x : xs) = do
                        tree <- genTree xs
                        let len = length xs
                        let totNodes = ((len * 2) - 1)
                        i <- pick 1 totNodes
                        replaceTreeAlg x i tree

--[1,2,3]
--tree <- genTree4 [2,3]
--        tree <- genTree[3]
--                L 3
--        tree =  L 3
--        len = 1
--        replaceTreeAlg 2 1 (L 3)
--        (B (L 2) (L 3))
-- tree = (B (L 2) (L 3))
-- len = 2
--replaceTreeAlg 1 2 (B (L 2) (L 3))
--   totNodes = 2 * 2 - 1 = 3
--   i = pick 1 3
--   i = 1
--   leftCount = 1
--   rightCount = 1
--



-- replaceTreeAlg :: PickingMonad m => a -> Int -> Bin a -> m (Bin a)
-- replaceTreeAlg val len (L x) = replaceTree val (L x)
-- replaceTreeAlg val len (B (left) (right)) = do
--                                                   let totNodes = ((len * 2) - 1)
--                                                   i <- pick 1 totNodes
--                                                   let leftCount = countNodes left
--                                                   let rightCount = countNodes
right
```

```
--                                                  if i <= leftCount then do
--                                                                          newLeft
<- replaceTreeAlg val (len - rightCount - 1) left
--                                                                          return
(B (newLeft) (right))
--                                                  else if i == (leftCount + 1)
then replaceTree val (B (left) (right))
--                                                  else do
--                                                          newRight <-
replaceTreeAlg val (len - leftCount - 1)  right
--                                                          return (B (left)
(newRight))


replaceTreeAlg :: PickingMonad m => a -> Int -> Bin a -> m (Bin a)
replaceTreeAlg val i (L x) = replaceTree val (L x)
replaceTreeAlg val i (B (left) (right)) = do
                                                 let leftCount = countNodes left
                                                 let rightCount = countNodes right
                                                 if i <= leftCount then do
                                                                        newLeft <-
replaceTreeAlg val i left
                                                                        return (B
(newLeft) (right))
                                                 else if i == (leftCount + 1) then
replaceTree val (B (left) (right))
                                                 else do
                                                         newRight <- replaceTreeAlg
val (i - leftCount - 1)  right
                                                         return (B (left)
(newRight))


replaceTree :: PickingMonad m => a -> Bin a -> m (Bin a)
replaceTree val tree = do
                        i <- pick 0 1
                        if i == 0 then return (B (L val) (tree))
                        else return (B (tree) (L val))

-- countNodes :: Int -> Int -> [Int] -> Bin a -> [Int]
-- countNodes count index array (B (L x) (L y)) = do
--                                                  replace (index - 1) 1
--                                                  replace (2 * index - 1) 0
--                                                  replace (2 * index) 0
-- countNodes count index array (B (left) (right)) = if (array !! index) == 0 then
B(left )


countNodes :: Bin a -> Int
countNodes (L x) = 1
countNodes (B (left) (right)) = countNodes left + countNodes right
```