99 questions/Solutions/21

From HaskellWiki

< 99 questions | Solutions

Insert an element at a given position into a list.

```
insertAt :: a -> [a] -> Int -> [a]
insertAt x xs (n+1) = let (ys,zs) = split xs n in ys++x:zs
```

or

```
insertAt :: a -> [a] -> Int -> [a]
insertAt x ys     1 = x:ys
insertAt x (y:ys) n = y:insertAt x ys (n-1)
```

There are two possible simple solutions. First we can use `split` from problem 17 (or even **`splitAt`** from the Prelude) to split the list and insert the element. Second we can define a recursive solution on our own.

As a note to the above solution - this presumes that the inserted argument will be a singleton type `a` inserted into a list `[a]`. The lisp example does not infer this intent. As a result, presuming the data to be inserted is likewise of type `[a]` (which we are tacitly inferring here to be String into String insertion), a solution is:

```
insertAt x xs n = take (n-1) xs ++ [x] ++ drop (n-1) xs
```

This solution, like many others in this quiz presumes counting element positions starts at 1, perhaps causing needless confusion.

A solution using foldl and a closure, also assumes lists are 1 indexed:

```
insertAt :: a -> [a] -> Int -> [a]
insertAt el lst n = fst $ foldl helper ([],1) lst
    where helper (acc,i) x = if i == n then (acc++[el,x],i+1) else (acc++[x],i+1)
```

The use of foldl imposes the use of concatenation. With a foldr we can use (:) instead, which is faster (O(n) vs. O(n^2)). The use of zip [1..] does not seem to add any overhead compared to the same solution with the index stored in the accumulator.

```
insertAt :: a -> [a] -> Int -> [a]
insertAt elt lst pos = foldr concat' [] $ zip [1..] lst
    where
        concat' (i, x) xs
            | i == pos  = elt:x:xs
```

```
        | otherwise = x:xs
```

Compared to the simple recursive definition, the fold version visits every elements of the list, whereas we could just stop after insertion of the element.

Category:

- Programming exercise spoilers
