Extended Example: Simple Tree Editor using the “Zipper”

“General” Trees:

- A forest is a list of tree nodes, each of which has a value and a forest of children:

```
type Forest a = [Node a]
data Node a  = Node a (Forest a)
```

- A simple example:

```
myForest     :: Forest String
myForest     = [Node "1"
                 [Node "1.1"
                   [Node "1.1.1" []],
                  Node "1.2" []],
                Node "2" []]
```

Operations on Forests:

- forestElems enumerates the values in a forest in depth-first order:

```
forestElems   :: Forest a -> [a]
forestElems   = concat . map nodeElems
  where nodeElems (Node x cs) = x : forestElems cs
```

- depthMap annotates a forest using depth information:

```
depthMap     :: (Int -> a -> b) -> Int -> Forest a -> Forest b
depthMap f d  = map depthNode
 where depthNode (Node x cs)
          = Node (f d x) (depthMap f (d+1) cs)
```

Displaying Forests:

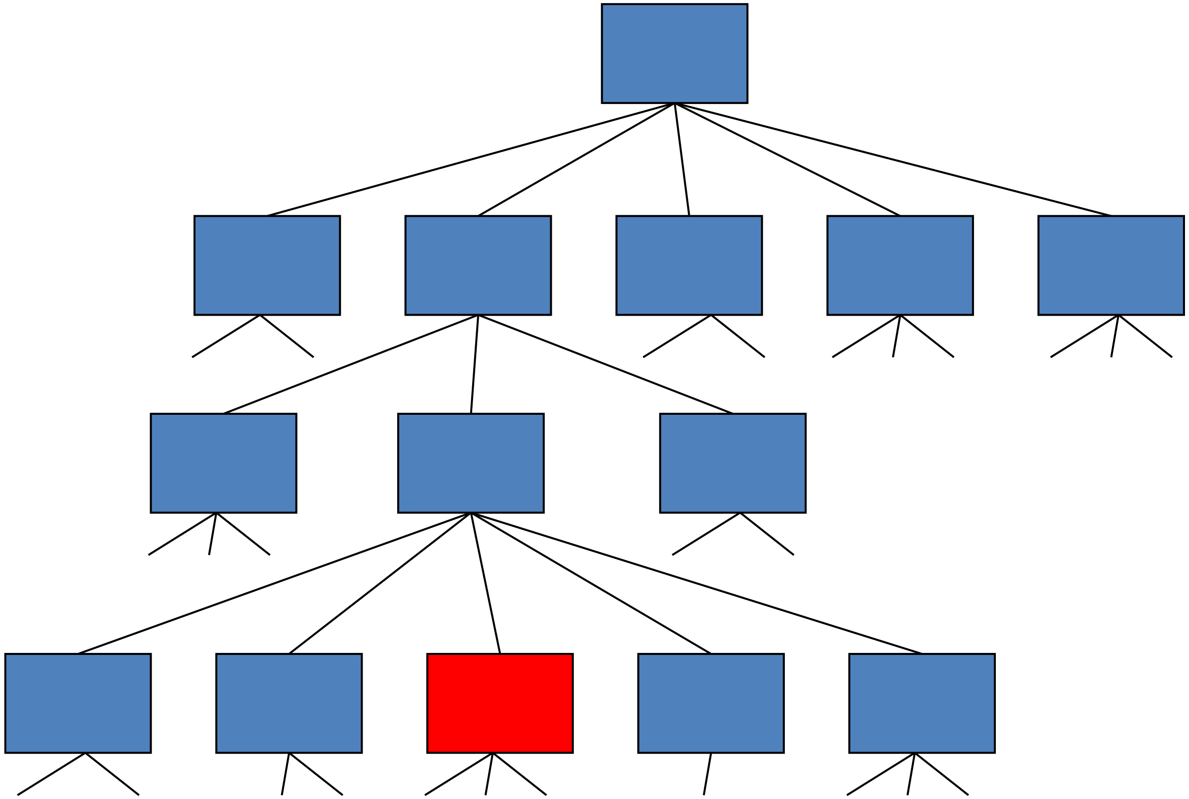
- Displaying a forest:

```
showForest          :: Forest String -> String
showForest          = unlines
                        .  forestElems
                        .  depthMap indent 1
  where indent d xs = replicate (2*d) '\SP' ++ xs
```

- Note: (from the Prelude)

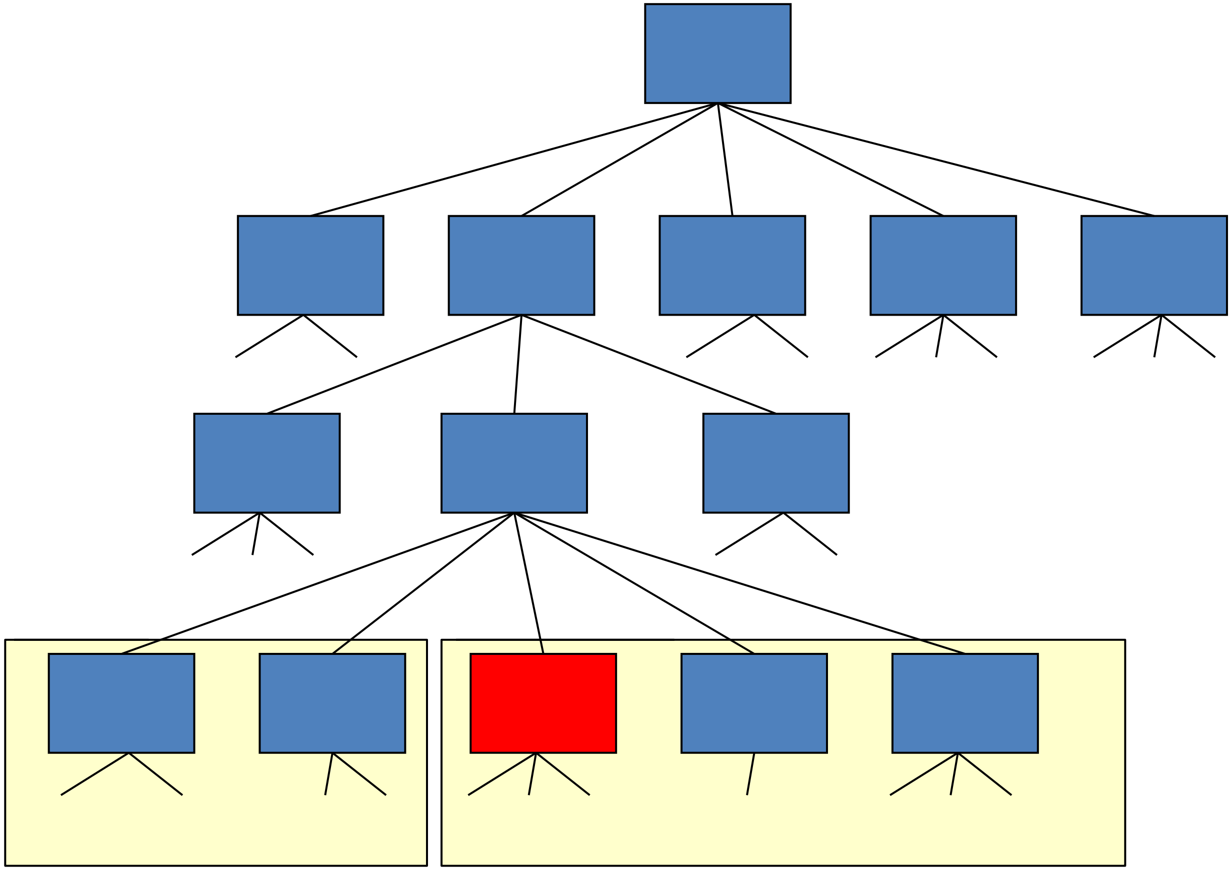
```
unlines  :: [String] -> String
unlines  = concat . map (++"\n")
```

Positions in a Tree:



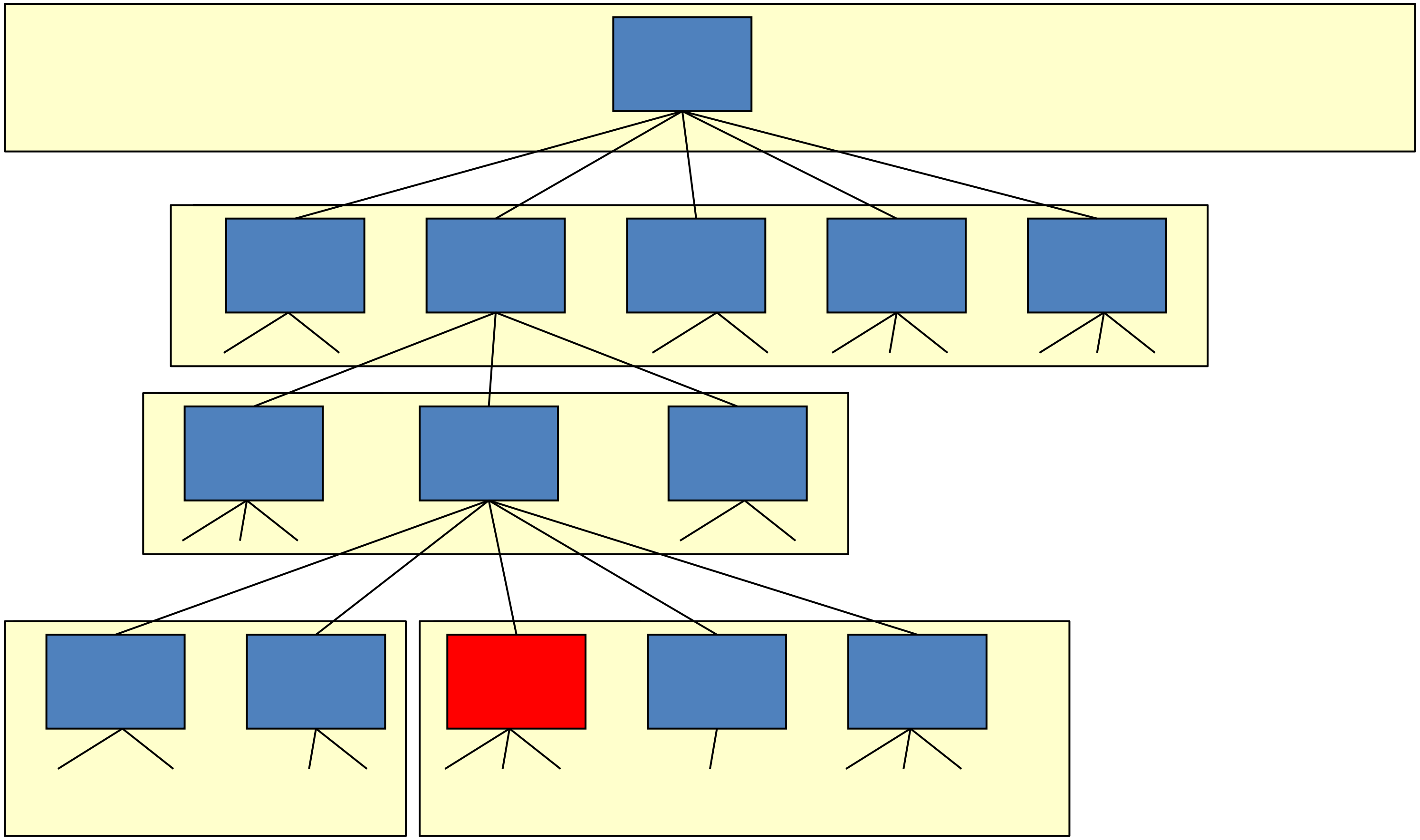
How can you identify a particular position in a tree … without pointers?

Positions in a Tree:



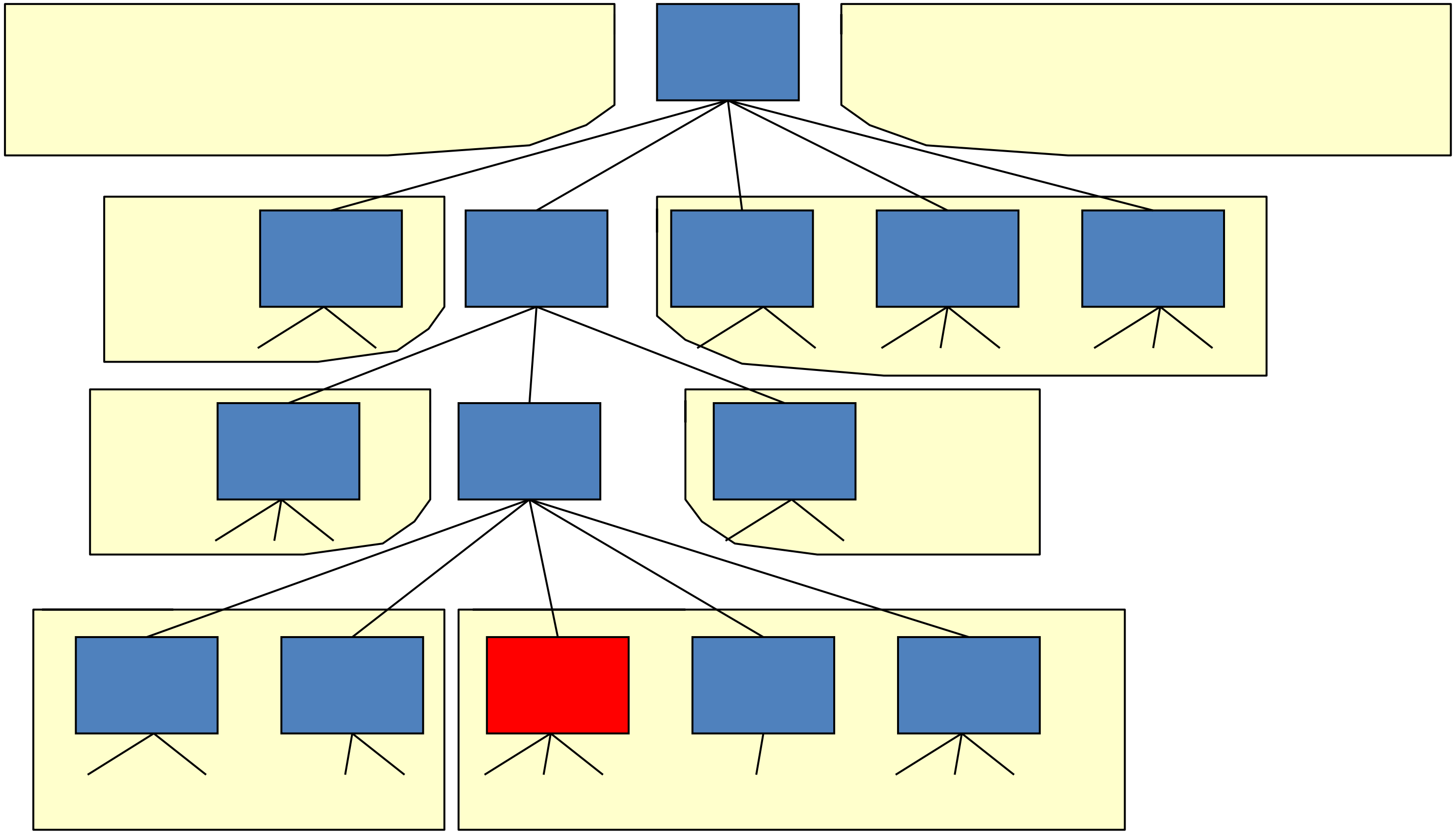
Split the row containing the current node into a left and right portion

Positions in a Tree:



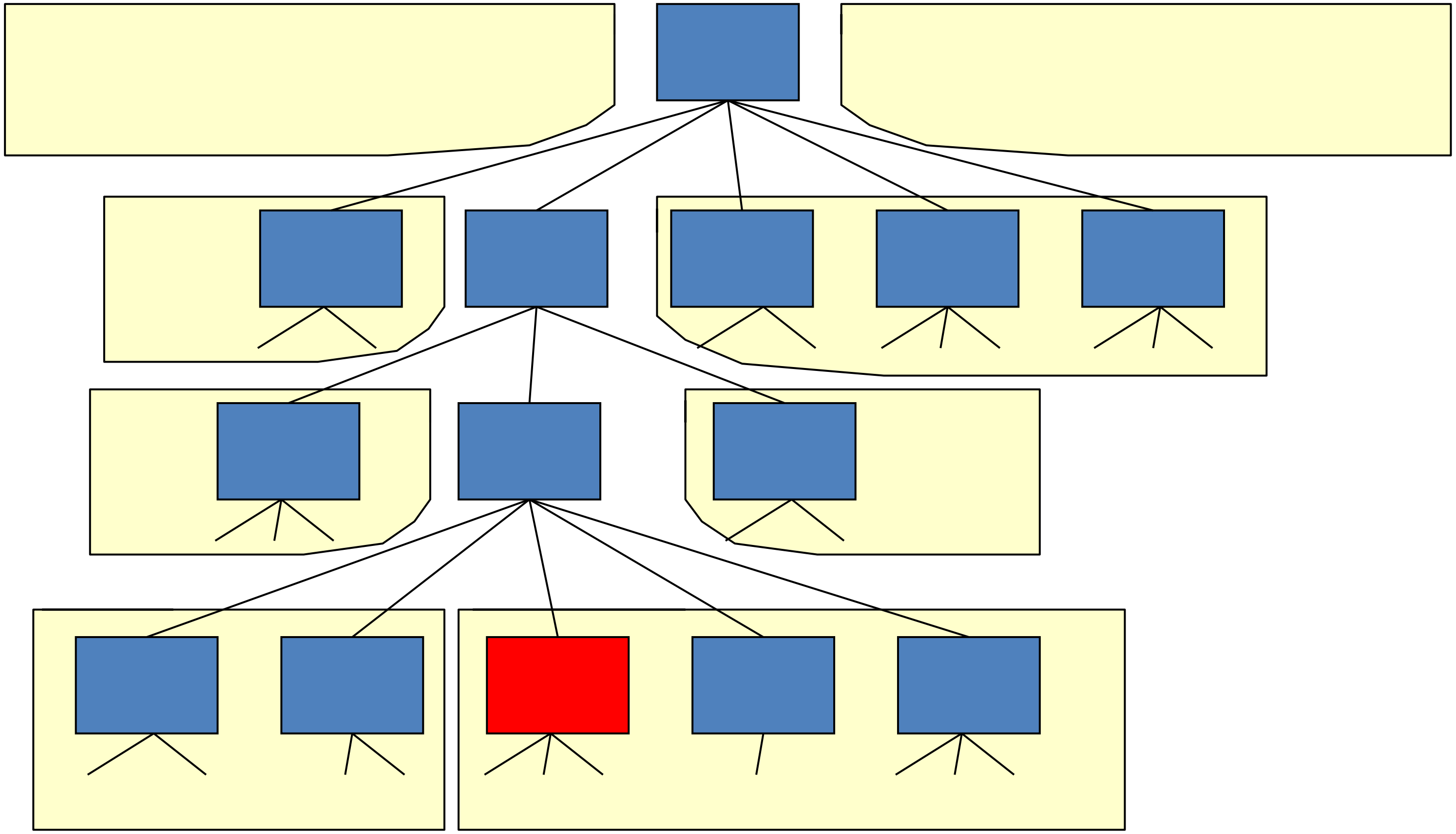
Add the layers on top

Positions in a Tree:



Where each layer contains a left portion, a single element, and a right portion

Positions in a Tree:



```
data Position a = Pos [Node a] [Level a] [Node a]
type Level a    = ([Node a], a, [Node a])
```

Forests and Positions:

Converting between forests and positions:

```
rootPosition    :: Forest a -> Position a
rootPosition f  = Pos [] [] f

reconstruct                   :: Position a -> Forest a
reconstruct (Pos ls us rs) = foldl recon (reverse ls ++ rs) us
 where recon fs (ls,x,rs) = reverse ls ++ [Node x fs] ++ rs
```

Note: reconstruct looses information

reconstruct . rootPosition = id

rootPosition . reconstruct ≠ id

Moving Around a Forest:

```
moveLeft, moveRight
      :: Position a -> Maybe (Position a)

moveLeft  (Pos ls us rs)
 = case ls of
          []      -> Nothing
          (n:ns) -> Just (Pos ns us (n:rs))

moveRight (Pos ls us rs)
 = case rs of
          []      -> Nothing
          (n:ns) -> Just (Pos (n:ls) us ns)
```

Identifying a Recurring Pattern:

```
repos        :: [b] -> (b -> [b] -> a) -> Maybe a
repos xs f = case xs of
                          []      -> Nothing
                          (n:ns) -> Just (f n ns)

moveLeft  (Pos ls us rs)
     = repos ls (\n ns -> Pos ns us (n:rs))

moveRight (Pos ls us rs)
    = repos rs (\n ns -> Pos (n:ls) us ns)

moveDown  (Pos ls us rs)
    = repos rs (\(Node x cs) ns ->
                                          Pos [] ((ls,x,ns):us) cs)
```

Other Operations:

- Modifying the tree:

```
insertNode  :: a -> Position a -> Position a
insertNode x (Pos ls us rs)
                   = Pos ls us (Node x [] : rs)

deleteNode :: Position a -> Maybe (Position a)
deleteNode (Pos ls us rs)
                   = repos rs (\_ ns -> Pos ls us ns)
```

- Reflecting the tree:

```
reflect       :: Position a -> Position a
reflect (Pos ls us rs) = Pos rs us ls
```

For Further Information:

- *A simple interactive tree editor*, Mark P Jones
- *Functional Pearl: The Zipper*, Gérard Huet