Understanding the Types

Here is a datatype to implement a BST.

```haskell
data BST a = Empty
           | Node a (BST a) (BST a)
deriving (Show, Eq)

data Box a = Box a

let t1 = Node 4 Empty (Node 5 Empty Empty)
let b1 = Box 20
```

**Problem 1)** For the BST type, what is the purpose of Empty and Node?

**Problem 2)** The text Box a occurs twice in the above code. What is the difference between the two occurrences?

**Problem 3)** The Haskell expression show t1 works, but show b1 does not. Why not?

**Problem 4)** Consider the following assignments for b2, t2, and t3. They are not legal. why not?

```haskell
let b2 = Box 10 20
let t2 = Node 8 (BST 3) (BST 4)
let t3 = Node "hi" Empty (Node 10 Empty Empty)
```

**Problem 5)** Consider the following helper functions. Two allow us to deal with leaf nodes, and one performs a left-rotation. There are quite a few references to Node and Empty here, but only a few of them cause memory to be allocated. Where are they, and how can you tell?

```haskell
isLeaf (Node x Empty Empty) = True
isLeaf _ = False
mkLeaf n = Node n Empty Empty
rotateLeft (Node b a (Node d c e)) = Node d (Node b a c) e
```
Implementing Add

**Problem 6)** Can you write the corresponding `rotateRight` function?

Consider this code:

```haskell
0 add elt Empty = mkLeaf elt
1 add elt (Node x a b) | elt < x = Node x (add elt a) b
2 | elt > x = Node x a (add elt b)
3 | otherwise = n
```

**Problem 7)** We haven't gone over the `n@` syntax yet. What do you think it means, and what would happen if we didn't have it?

**Problem 8)** How does this data structure handle it if we add multiple copies of the same element?

**Problem 9)** Write a function that will create a tree from the elements of a list. For extra Haskell points, do it in **one line** using a higher order function.

```haskell
Prelude> list2Tree [1,3,2]  
Node 2 (Node 1 Empty Empty) (Node 3 Empty Empty)
```
Implementing Delete

```clojure
0 del victim Empty = Empty
1 del victim (Node foo left right)
  | foo > victim = Node foo (delete victim left) right
  | foo < victim = Node foo left (delete victim right)
  | foo == victim =
2   case (left,right) of 
3    (Empty,Empty) -> Empty
```

**Problem 10)** What cases does the starter code above handle? Oh, and there's a bug; please fix that.

**Problem 11)** Extend the code to handle the case where there is one child.

**Problem 12)** Consider the following helper function.

```clojure
0 goLeft (Node a _ Empty) = a
1 goLeft (Node a _ b)     = goLeft b
```

How can this function be of use to us?

**Problem 13)** Implement two child deletion. Using `let`, you can actually do this in one or two lines.