Learning Objectives

1. ...

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)
```

**Problem 1** Consider the following assignments \( t_1 \), \( t_2 \), and \( t_3 \). Which are legal, and which are not? For the ones that are not, why not?

1. \( t_1 = \text{Node} 4 \text{ Empty} (\text{Node} 5 \text{ Empty Empty}) \)
2. \( t_2 = \text{Node} 8 (\text{BST} 3) (\text{BST} 4) \)
3. \( t_3 = \text{Node} "\text{hi}" \text{ Empty} (\text{Node} 10 \text{ Empty Empty}) \)

**Problem 2** 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 \text{Node} and \text{Empty} here, but only a few of them cause memory to be allocated. Where are they, and how can you tell?

1. \text{isLeaf} (\text{Node} x \text{ Empty Empty}) = \text{True}
2. \text{isLeaf}_\_ = \text{False}
3. \text{mkLeaf} n = \text{Node} n \text{ Empty Empty}
4. \text{rotateLeft} (\text{Node} b a (\text{Node} d c e)) = \text{Node} d (\text{Node} b a c) e

**Problem 3** Can you write the corresponding \text{rotateRight} function?
Implementing Add

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

**Problem 4** 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 5** How does this data structure handle it if we add multiple copies of an element?

**Problem 6** 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
1 Prelude> list2Tree [1,3,2]
2 Node 2 (Node 1 Empty Empty) (Node 2 Empty Empty)
```
Implementing Delete

```haskell
1 del victim Empty = Empty
2 del victim (Node foo left right)
3     | foo > victim = Node foo (delete victim left) right
4     | foo < victim = Node foo left (delete victim right)
5     | foo == victim =
6     case (left,right) of
7         (Empty,Empty) -> Empty
```

**Problem 7)** What cases does the starter code above handle? Oh, and there's a bug; please fix that.

**Problem 8)** Extend the code to handle the case where there is one child.

**Problem 9)** Consider the following helper function.

```haskell
1 goLeft (Node a _ Empty) = a
2 goLeft (Node a b) = goLeft b
```

How can this function be of use to us?

**Problem 10)** Implement two child deletion. Using `let`, you can actually do this in one or two lines.
Algebraic Data Types Activity--- Reflector's Report

<table>
<thead>
<tr>
<th>Role</th>
<th>Responsibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manager</td>
<td>Keeps team on track</td>
</tr>
<tr>
<td>Recorder</td>
<td>Records decisions</td>
</tr>
<tr>
<td>Reporter</td>
<td>Reports to Class</td>
</tr>
<tr>
<td>Reflector</td>
<td>Assesses team performance</td>
</tr>
</tbody>
</table>

1. What was a strength of your team's performance for this activity?

2. What could you do next time to increase your team's performance?

3. What insights did you have about the activity or your team's interaction today?