Development 8 - Exercises
Unit 5

For these exercises use the following type definitions:

BINARY TREE:

```
type BinaryTreeData<a> = {
  kind: "empty"
} | {
  kind: "node",
  value: a
  left: BinaryTree<a>,
  right: BinaryTree<a>
}
```

FUNCTOR:

```
interface Functor<F, G, a, b> {
  map: (this: F, f: (x: a) => b) => G
}
```

Moreover, try to express the operations on the data structures as immutable methods by extending the basic data structure with a record of functions.

Exercise 1:

Implement a function

```
let tryFind = <a>(value:  a) => (tree:  BinaryTree<a>):  Option<BinaryTree<a>>
```

that looks up for an element in a binary search tree. If the element is not found the function returns `None`.

Exercise 2:

Implement a function

```
let insert = <a>(value:  a) => (tree:  BinaryTree<a>):  BinaryTree<a>
```

that inserts a new element in a binary search tree.

Exercise 3:

Implement a function

```
let inorderFold = <a, state>(f:  (s:  state) => (x:  a) => state)
=> (init:  state) => (tree:  BinaryTree<a>):  state
```

that carries an accumulator through the in-order traversal of the binary search tree and updates its value by executing the function `f`.

Exercise 4:

Implement a function

```
let treeMap = <a, b>(f:  (x:  a) => b) => (tree:  BinaryTree<a>):  BinaryTree<b>
```

that applies the function `f` to each node in a binary search tree by performing the in-order traversal. The function outputs a tree containing the results of the function application. SUGGESTION: to preserve the binary search property use a fold on the tree to accumulate the result.

Exercise 5:

Extend the `Option` data type to be a functor using the function record `Functor<F, G, a, b>`

Exercise 6:

Extend the `List` data type to be a functor using the function record `Functor<F, G, a, b>`

Exercise 7:

Extend the `Tree` data type to be a functor using the function record `Functor<F, G, a, b>`