# Assignment 2 - BST Set Implementation

Rory Stephenson 300160212 stepherory

14 April 2014

## Module Setup

Internal helper functions are not exported, nor are the constructors of `Set`. Without the constructors the only way to obtain or modify a `Set` is via the exported functions which means that as long as these maintain the validity of the tree structure, an invalid `Set` cannot be obtained. This obviously excludes the case where the module is imported into ghci, ignoring what the module hides.

`Prelude` is explicitly imported in order to hide its implementation of `foldr`, `filter` and `map` (which collide with functions in this module). `Data.Foldable` is used to implement folds on the `Set`.

```
module Set (
  Set,
  add,
  remove,
  isEmpty,
  empty,
  singleton,
  member,
  size,
  isSubsetOf,
  union,
  intersection,
  difference,
  filter,
  valid,
  map) where

```

```
import Prelude hiding(foldr, filter, map)
import Data.Foldable
```

## Data and Instance declarations

`Set` is declared with two constructors, `EmptySet` and `Node`, which give it a tree structure. `EmptySet` acts as a leaf and is also used to represent a set with no elements in it. `Node` provides node structure in the tree, it holds an element as well as a left and right `Set`. The use of `a` in the set declaration ensures that the set can only hold one type.

The three instance declarations implement `Show`, `Foldable` and `Eq`. `Show` is used to render the set as a string in the format $\{\mathrm{e}_1, \mathrm{e}_2\ , .., \mathrm{e}_n\}$ where the elements are ordered. Whilst sets do not have order, the ordering reflects the structure of the underlying tree. `Foldable` allows `Set` to be folded on and also acts in order (to provide some predictability for someone folding on a `Set` with a function like (-) where order matters). The drawback of this method is that if a fold is used to add to a set, the resulting set may have the performance of a linked list because the elements will be added in order. Finally `Eq` checks equality of two sets by checking whether each set contains all of the elements of the other set.

```
data Set a = EmptySet | Node a (Set a) (Set a)


instance Show a => Show (Set a) where
    show a = "{" ++ drop 2 (foldr (\e r -> ", " ++ show e ++ r) "" a) ++ "}"

instance Foldable Set where
  foldr _ z EmptySet = z
  foldr f z (Node k l r) = foldr f (f k (foldr f z r)) l

instance Ord a => Eq (Set a) where
  EmptySet == EmptySet = True
  a == b = (foldr (\e r -> r && (member e b)) True a) &&
           (foldr (\e r -> r && (member e a)) True b)
```

`add` is a typical BST insert except it will not add the element if it is a duplicate (since this tree represents a set), it instead returns the tree unmodified. `add` recurses down the left/right branches of the tree if the new element is less/greater than the current node and returns a new node with no children once the appropriate position is found. Elements to be added must be an instance of `Ord` because finding the correct location of an element in the tree requires comparison.

```
add :: Ord a => a -> Set a -> Set a
add a EmptySet = Node a (EmptySet) (EmptySet)
add a (Node b c d) | a == b = Node b c d
                   | a < b = Node b (add a c) d
                   | otherwise = Node b c (add a d)
```

`remove` recurses down the tree until the specified element is found. If the element is in a node with one child it replaces the element with its child, otherwise it removes the node's in-order predecessor (found by calling `getRightChild` on the left child of the node) and replaces the node with the removed child. `Ord` is again requried in order to find the target node in the tree via comparisons.

```
remove :: Ord a => a -> Set a -> Set a
remove _ EmptySet = EmptySet
remove a n@(Node b EmptySet EmptySet) | a == b = EmptySet
                                      | otherwise = n
remove a (Node b EmptySet d) | a == b = d
remove a (Node b c EmptySet) | a == b = c
remove a (Node b c d) | a < b = Node b (remove a c) d
                      | a > b = Node b c (remove a d)
                      | otherwise =
                        Node (getRightChild c) (remove (getRightChild c) c) d
```

`getRightChild` recursively descends down the right branch of a node until it reaches the rightmost child and returns it. An error is thrown if `getRightChild` is called on `EmptySet` because a leaf has no children and no logical element could be returned which could be considered the right child of a leaf. The element also couldn't satisfy the function's type declaration, which requires an element of the same type of that which is held in the provided set to be returned, since the set holds no elements. This error should never occur during runtime because the function is only used in `remove` and it is called on the left child of a `Node` which has two children.

```
getRightChild :: Set a -> a
getRightChild (Node b _ EmptySet) = b
getRightChild (Node _ _ d) = getRightChild d
getRightChild EmptySet = error "Cannot find right child of an empty set."
```

`isEmpty` simply returns `True` if the provided set is the `EmptySet` or `False` otherwise.

```
isEmpty :: Set a -> Bool
isEmpty EmptySet = True
isEmpty _ = False
```

`empty` returns the `EmptySet`. `Ord` is not enforced here both because it is not required by the function and because it will be enforced by `add` if anything is added to the resulting empty `Set`. Having an order in an empty tree structure is meaningless.

```
empty :: Set a
empty = EmptySet
```

`singleton` returns a set containing only the specified element. Once again `Ord` is not enforced since it will be enforced by other functions if anything is added or removed. There is also no need for ordering in a tree that only has a root.

```
singleton :: a -> Set a
singleton a = Node a (EmptySet) (EmptySet)
```

`member` returns `True` when the given element is in the given `Set` or `False` otherwise. It recurses in a similar fashion to `add`, returning `True` if it finds an element which is equal to the provided one or `False` if it reaches `EmptySet`. `Ord` is needed for the tree navigation and to check if the desired element has been found.

```
member :: Ord a => a -> Set a -> Bool
member _ EmptySet = False
member a (Node b _ _) | a == b = True
member a (Node b c _) | a < b = member a c
member a (Node b _ d) | a > b = member a d
                      | otherwise = False
```

`size` calculates the number of elements in the `Set` by counting how many nodes the set has (since leaves cannot hold elements).

```
size :: Set a -> Int
size (EmptySet) = 0
size (Node _ b c) = 1 + size b + size c
```

`isSubsetOf` returns `True` if all the elements of the first set are contained in the second set (opposite to the assignment brief because it makes more sense when the function is called infix). `Ord` is required since member is called to check if it is a subset.

```
isSubsetOf :: Ord a => Set a -> Set a -> Bool
isSubsetOf EmptySet _ =  True
isSubsetOf s1 s2 = foldr (\e r -> r && (member e s2)) True s1
```

`union` returns a set containing all the elements in both sets (no duplicates). Using a fold to add the second set to the first means that the resulting set may not perform well if the two sets' elements are not of a similar range because all of the added elements will form a linked list beginning from the closest element in the other set. `Ord` is required because `add` is called inside this function on the provided sets.

```
union :: Ord a => Set a -> Set a -> Set a
union = foldr add
```

`intersection` creates a `Set` containing all the elements which are in both of the provided sets. It suffers from the same issue described in `union` but to an even greater degree because it adds all qualifying elements to a new empty `Set`, in order.

```
intersection :: Ord a => Set a -> Set a -> Set a
intersection s1 = foldr (\e r -> addIf (\ el -> member el s1 ) e r) EmptySet
```

`difference` returns a set containing the elements of the first `Set` which are not contained in the second `Set`. It suffers from the same issue as `intersection`.

```
difference :: Ord a => Set a -> Set a -> Set a
difference s1 s2 =
  foldr (\e r -> addIf (\ el -> not $  (member el s2)) e r) EmptySet s1
```

`filter` returns a `Set` containing all the elements which evaluate to `True` when the given function is applied to them. It suffers from the same issue as `intersection`.

```
filter :: Ord a => (a -> Bool) -> Set a -> Set a
filter f = foldr (\ e r -> addIf f e r) EmptySet
```

addif is an internal helper function which returns the specified set, adding the specified element to that set if the given function returns True when applied to that element.

```
addIf :: Ord a => (a -> Bool) -> a -> Set a -> Set a
addIf f a r | f a = add a r
            | otherwise = r
```

valid Evaluates whether a Set is valid both in terms of the semantics of sets and the underlying tree structure. Specifically it checks whether:

1. There are any duplicates in the set
2. Every Node's element is greater than those in its left subtree and less than those in its right subtree.

```
valid :: Ord a => Set a -> Bool
valid EmptySet = True
valid (Node _ EmptySet EmptySet) = True
valid (Node a (Node b c d) EmptySet) = a > b && valid (Node b c d) &&
                                       (not (member a c || member a d))
valid (Node a EmptySet (Node b c d)) = a < b && valid (Node b c d) &&
                                       (not (member a c || member a d))
valid (Node a l@(Node b c d) r@(Node e f g)) = a > b && a < e &&
                 valid (Node b c d) && -- The left child is correctly arranged
                 valid (Node e f g) && -- The right child is correctly arranged
                 not (member a l) && -- Element is not contained in left side
                 not (member a r) -- Element is not contained in right side
```

map implements mapping over a Set. It suffers from the linked list problem resulting from the use of folding.

```
map :: Ord b => (a -> b) -> Set a -> Set b
map f = foldr (\ e r -> add (f e) r) EmptySet
```