Abstract Data Type - Sets

Often times it is not clear how to represent the values of a data type. There may be multiple representations with associated pros and cons with regard to efficiency. The aim of *abstract data types* is to separate the data types from the representation so that a programmer using that data can work with it in a representation-independent way. This not only frees up the programmer from having to deal with unnecessary detail and complexity, it allows the representation to be changed without affecting how the data will be used. This representation-independent way of representing data is call *data abstraction* and the data is referred as an abstract data type or ADT.

We will illustrate abstract data types with the data type `Set`. Sets appear throughout mathematics and computer science and can be represented in numerous ways. Suppose a set contains elements of some type `a`. (Haskell allows us to leave the type unspecified when we define the type `Set`.) We start by defining the operations on sets which include:

Function	Explanation
`empty :: Set a`	gives an empty set
`isEmpty :: Set a -> Bool`	determines if a set is empty
`member :: Set a -> a -> Bool`	determines whether a value is contained in a set
`insert :: a -> Set a -> Set a`	adds a value to a set
`delete :: a -> Set a -> Set a`	deletes a value to a set

A simple representation of a set could be an unordered list of unique values. We can easily write the operations above but we would like to package them up so that a user of this type doesn't have to worry about the details. We will do this inside a module and export the operations. We will also use the `newtype` declaration to hide the construction of the data as a list.

```
module Set (Set, empty, isEmpty, member, insert, delete) where

newtype Set a = MakeSet([a])

empty :: Set a
empty = MakeSet([])

isEmpty :: Set a -> Bool
isEmpty (MakeSet x) = null x

member :: (Eq a) => Set a -> a -> Bool
member (MakeSet []) y = False
member (MakeSet(x : xs)) y = if (x == y) then True
                             else member (MakeSet xs) y
insert :: (Eq a) => a -> Set a -> Set a
insert x (MakeSet y) = if not (member(MakeSet y) x)then MakeSet (x : y)
                       else MakeSet y
```

Note: In `member` and `insert` we needed to specify that type `a` must have an equality test available.

What is the efficiency for each of the operations?

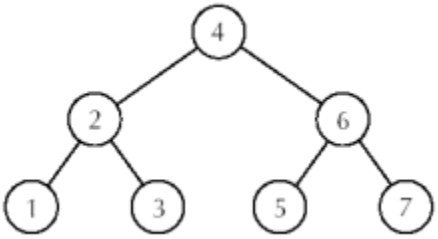
A user can now import this module and use the sets. For example,

```
module Example8 where
import Set

s1 = insert 3 (insert 5 (insert 1 empty))
s2 = insert 5 (insert 2 empty)
```

Here we have created two sets without regard to their implementation.

For the `member` function we had to search the list to find the value in question. Is there a way to make this search more efficient? We could represent the data as a tree. We can order the elements in the tree such that everything to the right of the root is smaller than the root, and everything to the left is larger. The two subtrees can also have this property. Such a tree is called a *binary search tree*. Here is an example:



Now when we do a search, the time it takes will be, at worst, the depth of the tree. If a tree has n elements and is balanced, what would be the depth of the tree?

Here is a representation of sets using binary search trees:

```
module Set (Set, empty, isEmpty, member, insert, delete) where

data Bst a = Null | Branch a (Bst a) (Bst a)
type Set a = Bst a

empty :: Set a
empty = Null

isEmpty :: Set a -> Bool
isEmpty Null = True
isEmpty (Branch x t1 t2) = False
```

```
member :: (Ord a) => Set a -> a -> Bool
member Null y = False
member (Branch x t1 t2) y =  if (y < x) then
      member t1 y
else if (x == y) then
      True
else member t2 y

insert :: (Ord a) => a -> Set a -> Set a
insert x Null = Branch x Null Null
insert x (Branch y t1 t2) = if (x < y) then
     Branch y (insert x t1) t2
else if (x == y) then
     Branch y t1 t2
else
     Branch y t1 (insert x t2)

delete :: (Ord a) => a -> Set a -> Set a
delete x Null = Null
delete x (Branch y t1 t2) = if (x < y) then
     Branch y (delete x t1) t2
else if (x == y) then
     join t1 t2
else
     Branch y t1 (delete x t2)

join :: Set a -> Set a -> Set a
join t1 t2 = if isEmpty t2 then t1
     else Branch y t1 t3
     where (y,t3) = splitTree t2

splitTree :: Set a -> (a, Set a)
splitTree (Branch y t1 t2) = if isEmpty t1 then (y,t2)
     else (u, Branch y t3 t2)
     where (u,t3) = splitTree t1
```

Note that we needed to specify that we can test order relations on type `a`. We also wrote auxiliary functions to perform the delete but these functions are not exported from the module. They are therefore hidden from the user.

The user would use this new representation in the identical way as before! This is the beauty of abstraction.

What is the efficiency of each of the operations with the binary search tree representation?