A TriTree is defined recursively as:

1. EmptyNode: An empty node
2. TriNode a: A node holding an element with three branches, each a trinary tree

Define this in Haskell. Your data type should be called **TriTree**, your two node types should be called **EmptyNode** and **TriNode**. In addition, your **TriNode** should contain any type (first) followed by the three possible branches. *Note: If you don't follow this format, we won't be able to grade your problem and it will just be marked wrong.* Add the following typeclass extension for your TriTree:

```
instance (Eq a) => Eq (TriTree a) where
 EmptyNode           == EmptyNode = True
 TriNode a la ma ra  == TriNode b lb mb rb = (a == b) &&
                    (la == lb) &&
                    (ma == mb) &&
                    (ra == rb)
 _                   == _ = False
```


This will allow us to annotate some of the grading for your solutions. Now, using this definition, write the following functions.

1. **nodeValue :: TriTree a -> a.** This function takes a trinary tree and returns the value of the given node. This should return an error if you attempt to do this on an empty node.
2. **leftChild :: TriTree a -> TriTree a.** Takes a trinary tree and returns the left child, or an error if the tree is an empty node.
3. **middleChild :: TriTree a -> TriTree a**. Takes a trinary tree and returns the middle child, or an error of the tree is an empty node.
4. **rightChild :: TriTree a -> TriTree a.** Takes a trinary tree and returns the right child, or an error if the tree is an empty node.
5. **inTree :: Eq a => a -> TriTree a -> Bool.** Returns True or False if the given element is in a TrinaryTree. For example, assuming **t** contains a trinary tree, inTree 5 t returns True if 5 is in it, or False otherwise. You can't assume the tree is sorted.
6. **leafList :: TriTree a -> [a].** Takes a trinary tree and returns a list of all the values in the leaves of the tree. A leaf is a node with three empty branches.
7. **inOrderMap :: (a -> b) -> TriTree a -> TriTree b.** This function acts like map on a list, performing an in-order traversal of the tree. Thus, you pass a function that takes type a and

returns type b, pass in a TrinaryTree with type a, and get a resulting TrinaryTree of type b. The trees will match in structure, but with each value transformed by the function you used.

8. **preOrderFold:: (b -> a -> b) -> b -> TriTree a -> b.** Similar to a foldl operation on lists, this function takes a function as the first argument, an accumulator value, a TriTree, and performs an pre-order walk of the tree, applying the function to each value and then using the result of that function in the next call of the folding in the tree. For example, a tree with (1 2 4 3), where 2, 4, and 3 are the children and then a call to preOrderFold (+) 0 (1 2 4 3), would result in 10. *Ask questions if this isn't clear.* Again, you can't do fancy things like derive from Foldable (it won't work as expected anyways).

1. **Pathfinding :** When you execute a find on a TriTree, you take some path to get there. Write a function called **findPath** that returns a list of paths taken to reach that location. For example, your list might be [PathLeft, PathRight, PathRight] if you first went down the left branch of the root node, then took two right branches to arrive at your destination. You should use this type for it: **data TreePath = PathLeft | PathMiddle | PathRight**. *Hint: You will want a helper function that maybe returns two values, wether the path was found and what it was up to that point.*
2. **findAndPrune :** Write a function similar to find above, but that prunes the branch where the value was found. This function returns a tree, but just with the branch (and subtree) removed.
3. **Pruning :** Write a function called **pruneWithPath** that duplicates a tree, but allows you to prune a particular branch and its subtree. The resulting tree should be everything but the pruned branch. This is similar to the previous problem, but instead of finding a node by its value, you must find it by a path, thus if you're given [PathLeft, PathRight, PathRight], you must follow that path and then prune.
4. **kTree :** Define a data type for k-trees, i.e., a tree with **k** children. A k-tree is similar to a binary tree, except that only the leaves have values, internal nodes do not. Implement **kLeafList, kMap,** and **kFold** for the kTree. To be clear, a kTree is either a Leaf (with a value) or a Node with k children so use the names: **KTree** for the type name, **KLeaf** for the leaves and **KNode** for an individual node.


```
data TriTree a = EmptyNode | TriNode a (TriTree a) (TriTree a) (TriTree a)

instance (Eq a) => Eq (TriTree a) where

  EmptyNode == EmptyNode = True
```

```
TriNode a la ma ra  == TriNode b lb mb rb = (a == b) &&
                                            (la == lb) &&
                                            (ma == mb) &&
                                            (ra == rb)
_ == _ = False
```