Implement a Linked List structure in Haskell according to the instructions below. Provide comments (including file headers with your name, date, homework number, etc.) and tests showing your functions work correctly. Turn in your assignment in class by the due date.

**Instructions.** Define a linked-list data type in Haskell called `List` that consists of value constructors `Node` and `Nil`. Your `List` type should be an instance of both `Eq` and `Show`. Implement the following functions for `List`:

1. `insert :: a -> List a -> List a`, which given a value and a list, returns a new list with the value inserted at the front of the list.
2. `delete :: (Eq a) => a -> List a -> List a`, which given a value and a list, returns a new list with the first occurrence of the value removed from the list.
3. `memberOf :: (Eq a) => a -> List a -> Bool`, which returns true if the given value is in the given list.
4. `elementAt :: Int -> List a -> a` which takes an index `i` and a list, and returns the `i`-th element in the list. The first element in the list should be element 0. If `i` is not a valid index, then `elementAt` should print the error “Index out of bounds”.
5. `insertAt :: Int -> a -> List a -> List a` which takes a value, an index `i`, and a list, and returns the list with the value inserted into the `i`-th position of the list (moving the current element at `i` to the right). If the index is invalid, `insertAt` should print the error “Index out of bounds”.
6. `deleteAt :: Int -> List a -> List a`, which takes an index `i` and a list, and returns the list with the `i`-th element removed. If the index is invalid, `deleteAt` should print the error “Index out of bounds”.
7. `appendWith :: List a -> List a -> List a`, which returns the result of appending the first list with the second list.
8. `reverseList :: List a -> List a`, which returns a list in the reverse order of the given list. You can call your `appendWith` function in your `reverseList` definition.
9. `zipList :: List a -> List b -> List (a,b)`, which mimics the `zip` function on the two given lists.
10. `mapList :: (a -> b) -> List a -> List b`, which mimics the `map` function on a list (i.e., it takes a function and applies it to each element of the list to obtain the resulting list).