Homework 4: Functional Data Structures

In this assignment, you'll build two data structures that use functional programming ideas in a unique way.

The template and data for this assignment are available here:
https://www.cs.umass.edu/~arjun/courses/cmpsci220-spring2016/hw/fundata.zip

1 Persistent Queues

Recall from earlier classes, that a *queue* is a data structure that supports three operations:

1. *Empty* constructs an empty queue,
2. *Enqueue* adds a new element to the back of the queue,
3. *Dequeue* removes an element from the front of the queue, if the queue is not empty.

In the following exercises, you will build a *persistent queue*. A persistent queue has the operations defined. But, instead of having enqueue and dequeue update the queue, they leave the original queue unchanged and return a new queue.

It is easy to implement a persistent queue using a list:

```
type SlowQueue[A] = List[A]

def emptySlow[A](): SlowQueue[A] = Nil()

def enqueueSlow[A](elt: A, q: SlowQueue[A]): SlowQueue[A] = q match {
  case Nil => List(elt)
  case head :: tail => head :: enqueueSlow(elt, tail)
}

def dequeueSlow[A](q: SlowQueue[A]): Option[(A, SlowQueue[A])] = q match {
  case Nil => None()
  case head :: tail => Some((head, tail))
}
```

Read the code above carefully. The *enqueue* operation traverses the entire list each time (i.e., $O(n)$ running time). Your task is to implement the queue more efficiently.

The trick is to represent the queue using two lists. The first list, called *front*, has the elements at the front of the queue. The second list, called *back*, has the elements at the back of the queue, *in reverse order*.

For example, if *front* is `List(1, 2, 3)` and *back* is `List(6, 5, 4)`, then the elements of the queue, in order, are 1, 2, 3, 4, 5, 6. With this representation:

- *Enqueue* adds an element to *back*, but doesn't need to traverse the whole list.
- *Dequeue* removes an element from *front*, unless *front* is empty. If it is empty, it reverses *back* and uses it as the front.

In the assignment template, the file `src/main/scala/Types.scala` defines a type called `Queue` that you should use to define define the following functions:

```
def enqueue[A](elt: A, q: Queue[A]): Queue[A]

def dequeue[A](q: Queue[A]): Option[(A, Queue[A])]
```

2 Join Lists

A *join list* is a data structure that represents a list, but the elements are arranged into a tree. This tree-shape makes some operations, like list-concatenation very efficient. You'll be working with the `JoinList[A]` type, which is defined in `src/main/scala/Types.scala`. The type has three constructors:

1. `Empty()` represents an empty list.
2. `Singleton(x)` represents a list with one element x.
3. `Join(lst1, lst2, length)` represents `lst1` appended to `lst2`. The `length` field is the total number of elements in the list.

It should be clear that it is very cheap to append two join lists: you simply use the `Join` constructor. It is also cheap to calculate the length of a join list, since it is stored at each node.

Finally, since join lists represent lists, we've provided two functions to convert between join lists and tests in the `src/main/test/scala/Tests.scala` file:

- `toList[A](lst: JoinList[A]): List[A]` converts a join list into a Scala list. This operation can be very expensive, but is useful for testing.
- `fromList[A](lst: List[A]): JoinList[A]` converts a Scala list into a join list by repeatedly splitting a list into two equal halves.

These two fucntions are provided for testing only. You must not use them in your solution.

Programming Task Your task is to write some typically list-processing functions for join lists.

1. `max(lst, compare)` produces the maximum value in `lst`. The second argument is a comparator. If `compare(x, y) == true`, then `x` is greater than `y`. If the list is empty, the function produces `None`.
2. `map(f, lst)` produces a new join list, which has exactly the same shape as `lst`, but with `f` applied to every element.

3. `filter(pred, lst)` produces a new join list that has only includes elements of `lst` that satisfy the given predicate. The order of elements should not change.

4. `first(lst)` and `rest(lst)` produce the head and tail, respectively, of `lst` if it is non-empty.

5. `nth(lst, i)` produces the nth element of the list (the first element has index 0).

3 Hand In

From `sbt`, run the command `submit`. The command will create a file called `submission.tar.gz` in your assignment directory. Submit this file using Moodle.

For example, if the command runs successfully, you will see output similar to this:

```
Created submission.tar.gz. Upload this file to Moodle.
[success] Total time: 0 s, completed Jan 17, 2016 12:55:55 PM
```

Note: The command will not allow you to submit code that does not compile. If your code doesn't compile, you will receive no credit for the assignment.