Homework 4: Functional Data Structures

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

Preliminaries

You should create a directory-tree that looks like this:

```
./fundata
├── project
│   └── plugins.sbt
└── src
    ├── main
    │   └── scala. ............................................ Your solution goes here
    └── test
        └── scala............................................... Yours tests go here
```

The `project/plugins.sbt` file must have exactly this line:

```
addSbtPlugin("edu.umass.cs" % "compsci220" % "1.0.1")
```

You should implement your solution in a single file, within an object called `FunctionalDataStructures`.

Persistent Queues

Recall from earlier courses 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 above. 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:

```scala
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*.

```scala
object FunctionalDataStructures {

  case class Queue[A](front: List[A], back: List[A])

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

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

  sealed trait JoinList[A] {
    val size: Int
  }

  case class Empty[A]() extends JoinList[A] {
    val size = 0
  }

  case class Singleton[A](elt: A) extends JoinList[A] {
    val size = 1
  }

  case class Join[A](lst1: JoinList[A], lst2: JoinList[A], size: Int) extends JoinList[A]

  def max[A](lst: JoinList[A], compare: (A, A) => Boolean): Option[A] = ???

  def first[A](lst: JoinList[A]): Option[A] = ???

  def rest[A](lst: JoinList[A]): Option[JoinList[A]] = ???

  def nth[A](lst: JoinList[A], n: Int): Option[A] = ???

  def map[A,B](f: A => B, lst: JoinList[A]): JoinList[B] = ???

  def filter[A](pred: A => Boolean, lst: JoinList[A]): JoinList[A] = ???

}
```

Figure 13.1: Solution template.

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 your solution, use the following type to represent queues:

```
case class Queue[A](front: List[A], back: List[A])
```

Then, implement the following functions:

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

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

Join Lists

A *join list* is a data structure that represents a list, but the elements are arranged into a tree, as follows:

```
sealed trait JoinList[A] { val size: Int }
case class Empty[A]() extends JoinList[A] { val size = 0 }
case class Singleton[A](elt: A) extends JoinList[A] { val size = 1 }
case class Join[A](alist: JoinList[A], blist: JoinList[A], size: Int) extends JoinList[A]
```

This tree shape makes some operations, like list-concatenation, very efficient. The type has three constructors:

1. `Empty()` represents an empty list.
2. `Singleton(x)` represents a list with one element x.
3. `Join(alist, blist, length)` represents `alist` appended to `blist`. 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, it is easy to convert between join lists and normal lists.

- `toList[A](alist: 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](alist: List[A]): JoinList[A]` converts a Scala list into a join list by repeatedly splitting a list into two equal halves.

We've provided functions to do so in fig. 13.2. You may use these functions in your test suites. You must not use them in your solution.

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

1. `max(alist, compare)` produces the maximum value in `alist`. 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, alist)` produces a new join list, which has exactly the same shape as `alist`, but with `f` applied to every element.
3. `filter(pred, alist)` produces a new join list that has only includes elements of `alist` that satisfy the given predicate. The order of elements should not change.
4. `first(alist)` and `rest(alist)` produce the head and tail, respectively, of `alist` if it is non-empty.
5. `nth(alist, i)` produces the nth element of the list (the first element has index 0).

```
class Tests extends org.scalatest.FunSuite {
  import FunctionalDataStructures._

  def fromList[A](alist: List[A]): JoinList[A] = alist match {
    case Nil => Empty()
    case List(x) => Singleton(x)
    case _  => {
      val len = alist.length
      val (lhs, rhs) = lst.splitAt(len / 2)
      Join(fromList(lhs), fromList(rhs), len)
    }
  }

  def toList[A](alist: JoinList[A]): List[A] = lst match {
    case Empty() => Nil
    case Singleton(x) => List(x)
    case Join(alist1, alist2, _) => toList(alist1) ++ toList(alist2)
  }
}
```

Figure 13.2: Helpers for writing tests.

Hand In

From the `sbt` console, 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.