# HASKELL PROGRAMMING PROBLEM SET 4

LENNART JANSSON AND BRANDON AZAD

## Skew heaps

In the next few problems you will implement a skew heap data type, a simple, unstructured, breed of heaps well-adapted to Haskell that can be used to implement priority queues with every operation in at most $O(\log n)$ time. Cool!

A skew heap is built like a binary tree on the inside, where the heap property is respected. This means that every element is smaller than (or equal to) both of its children, and every element is larger than (or equal to) it's parent. The top of the heap is therefore the smallest element in the whole heap. Almost every operation on the skew heap is defined in terms of the union function, which combines two heaps into one and has type `union :: Ord a => SkewHeap a -> SkewHeap a -> SkewHeap a`. Therefore, we'll implement that first.

First, let's get used to the data type.

```
data SkewHeap a = Empty | SkewNode a (SkewHeap a) (SkewHeap a)
```

The way to parse this type: a `SkewHeap` that contains elements of type `a` is either `Empty`, or is a `SkewHeap` containing a thing of type `a` at the root, then a left subtree `SkewHeap a`, then a right subtree `SkewHeap a`. That's it!

### Problem 1. Simple things

Implement `empty :: SkewHeap a`. This should give a skew heap with no elements in it.

Implement `singleton :: a -> SkewHeap a`. This should take a single element and put it in a skew heap with just that one element.

Implement `null :: SkewHeap a -> Bool`, which should return `True` if the skew heap is completely empty, and `False` if it does contain some elements.

### Problem 2. Union

To take the union of two heaps, we find the heap with smaller root element, call it $t_a$ and the other $t_b$. Then, we take the union of $t_b$ and the right child of $t_a$, and that becomes the new left child of $t_a$. Then the original left child of $t_a$ becomes the new right child of $t_a$. Now $t_a$ contains all the elements from both $t_a$ and $t_b$, and the heap property is still respected, so the new $t_a$ is the union of the two heaps and is returned.

Implement `union :: Ord a => SkewHeap a -> SkewHeap a -> SkewHeap a`.

Also, define a new operator, for example `(+*+)`, that is an infix synonym for `union`. This will make your code prettier down the line since union will be very useful for the following functions.

### Problem 3. Insert and Extract

Implement `insert :: Ord a => a -> SkewHeap a -> SkewHeap a`. This takes an element and a skew heap and returns the skew heap with that element added.

Implement `extractMin :: Ord a => SkewHeap a -> (a, SkewHeap a)`, which takes a skew heap, and returns a tuple of the minimum element in the skew heap as well as the modified skew heap without that element.

**Problem 4. List interface**

Implement `fromList :: Ord a => [a] -> SkewHeap a`, which takes a list (not necessarily in sorted order) and puts the elements in a skew heap (where the heap property is respected).

Implement `toAscList :: Ord a => SkewHeap a -> [a]`, which takes a skew heap and gives a list of all the elements in the heap in ascending order.

Congratulations! You've implemented a new container type with a similar interface to `Data.Set` and `Data.Map`, and a not-too-shabby amount of code. Hopefully this gives you some idea about what goes on under the surface when you use container type modules: it's not mysterious stuff, just plain old Haskell you could understand or implement yourself.

As a reward...

**Problem 5. Heap sort**

Use skew heaps to implement `heapsort :: Ord a => [a] -> [a]` :)