CS 115: Functional Programming, Spring 2016

Assignment 2: Algebraic datatypes and modules

Due: *Monday, April 25, 02:00:00*

Coverage

This assignment covers the material up to lecture 7.

Submitting your assignment

Write the code for the priority queue implementation in a file called `PriorityQueue.hs`. Write the code for the heapsort implementation in a file called `HeapSort.hs`. Both files should be submitted to csman.

Writing your code

All of your Haskell functions, even internal ones (functions defined inside other functions), should have explicit type signatures.

All of your code should be tested using `ghci` with the `-W` (warnings enabled) command-line option. Any warnings will be considered errors, so make sure your code does not give any warnings.

Both files you will submit will be modules, which means they have to begin with a module declaration of the form:

```
module MODULENAME(EXPORTLIST) where
```

You will substitute the name of the module for `MODULENAME` and put the list of explicitly exported names in place of `EXPORTLIST`. If everything from the module is exported, write it like this:

```
module MODULENAME where
```

Below we will describe exactly which functions we want exported from each module.

As usual, at the beginning of each problem is an estimate of the maximum time in minutes it should take you to solve the problem (in boldface).

Testing your code

For this week's assignment, we are supplying you with a test script which you should download. You can

run it by putting it into the same directory as your two code modules, `cd`ing to that directory and typing:

```
% ghci Lab2Tests.hs
```

(where `%` is the terminal prompt, of course). Once inside `ghci`, type `main` at the prompt and the tests will be run.

This test script will run a number of tests on your code; any failure indicates that you have some debugging to do. The test script relies on two Haskell packages called `HUnit` and `QuickCheck`. If either of these packages aren't installed, install them by starting up a terminal and entering these commands:

```
% cabal install hunit
% cabal install quickcheck
```

If the packages are already installed, `cabal` will tell you and you shouldn't reinstall them.

`HUnit` is a fairly standard unit testing framework that tests whether some function applied to some arguments returns the expected results. `QuickCheck` is a randomized testing framework that tests that functions obey particular properties, automatically synthesizing large numbers of random data values to test the functions. (If you think this is pretty cool, you'd be right.)

About this assignment

The purpose of this assignment is to get you comfortable working with algebraic datatypes in Haskell, as well as to give you an understanding of the Haskell module system. To do this, you will be implementing a data structure called a priority queue and using it to implement a sorting algorithm called heapsort. The specific implementation of the priority queue you will create is called a leftist heap. So let's discuss these things now.

Priority queues

A priority queue is a data structure which stores a collection of items of a single type (where the type has to be orderable). The queue is ordered so that it provides fast access to its minimum element, as defined by the ordering relationship on the type of the items. Conceptually, you can think of the minimum element as "the next thing to be processed". The priority queue also has to have an operation which deletes the minimum element (returning a valid priority queue composed of the remaining elements), as well as several other operations. These can be represented by Haskell datatypes, values and functions with the following types:

```
-- Datatype for priority queues, parameterized around an element type a.
-- The element type should be an instance of the Ord type class.
data Pqueue a  -- implementation left out

-- An empty priority queue storing values of type a.
empty :: Pqueue a

-- Return True if the queue is empty.
isEmpty :: Pqueue a -> Bool

-- Insert an item into a priority queue.
insert :: Ord a => a -> Pqueue a -> Pqueue a
```

```
-- Find the minimum-valued element in a priority queue if possible.
findMin :: Ord a => Pqueue a -> Maybe a

-- Delete the minimum element from a priority queue if possible.
deleteMin :: Ord a => Pqueue a -> Maybe (Pqueue a)

-- Remove the minimum element if possible and return it,
-- along with the rest of the priority queue.
popMin :: Ord a => Pqueue a -> Maybe (a, Pqueue a)

-- Convert an unordered list into a priority queue.
fromList :: Ord a => [a] -> Pqueue a

-- Validate the internal structure of the priority queue.
isValid :: Ord a => Pqueue a -> Bool
```

A few notes about this:

- Some of the functions return `Maybe` types. If an empty priority queue is given as an argument to the `findMin`, `deleteMin`, or `popMin` functions, the functions return `Nothing`.
- The `fromList` function is not fundamental to the definition of a priority queue, but it's extremely useful.
- The `isValid` function returns `True` if the internal structure of the datatype (its internal invariants) is correct. This is used for testing and is not fundamental to the definition of a priority queue.
- We are being loose about the words "insert" and "delete". These operations are purely functional, so that when we say we "insert" an element into a queue, we really mean that we create a new queue which has the extra element in it; the original queue is unchanged. Similarly, when we "delete" an element from a queue, we really mean that we create a new queue without the offending element; the original queue is once again unchanged. That's functional programming for you.

Leftist heaps

Now that we know how our data structure is supposed to behave, the next question is: how do we implement it? Naturally, there are lots of choices, each of which will have different trade-offs. In this assignment you're going to implement priority queues as *leftist heaps*. This is a data structure that has the following attributes:

- It can either be a leaf (representing an empty heap and containing no data) or a node which contains the following items:
  - a stored element
  - a positive integer value called the "rank" of the node
  - a left and right subheap, each of which is also a leftist heap

  Thus, a leftist heap is a binary tree with additional rank information stored in the nodes. The tree will in general **not** be balanced; it will usually have more elements in the left subtree than in the right. We say that the tree "skews to the left".

Note that leftist heaps are very simply defined as a Haskell algebraic datatype. Such a definition is much simpler and closer to the verbal description of the datatype than a definition in terms of (for example) classes in an object-oriented language. This is one reason why algebraic datatypes are such a useful language feature.

The one thing to watch out for is that the definition of the priority queue datatype must be parameterized on a type variable, so it must look something like this:

```
data Pqueue a =
  -- code involving (Pqueue a) when referring to subheaps
```

Also note that even though the type variable `a` must be an instance of the `Ord` type class, we don't put that constraint into the datatype definition itself. Instead, all the functions that act on priority queues and that need the ordering property of type `a` must have explicit `Ord a =>` constraints in their type signatures, as you can see in the type signatures given above.

- The element stored at the top of the tree is smaller (more precisely, is no larger) than any of the elements stored beneath it in either subtree.
- As mentioned, each node has a positive integer value associated with it, which is called its *rank*. The rank of a node is equal to the length of its *right spine*, which is the number of nodes in the rightmost path from the node in question to an empty (leaf) node (this will also turn out to be the shortest path to a leaf node). Thus, a leaf node has rank 0, and a heap with a right subheap which is a leaf has rank 1. However, ranks are only stored in nodes, not in leaves. Having the rank stored in the nodes makes many operations much faster (the alternative would be to compute the rank on the fly, which would be very expensive for large heaps).
- The rank of any left child is required to be at least as large as the rank of its right sibling. This is why it's called a "leftist" heap.

The interesting feature of leftist heaps is that they support very efficient merging of two heaps to form a combined heap. (Of course, since we're using purely functional code, you don't alter the original heaps when merging.) Leftist heaps can be merged in O(log N) time, where N is the total number of elements in the resulting heap. Furthermore, once the merge operation (which, you'll note, is not a required operation for priority queues) is implemented, you can define most of the rest of the operations very easily in terms of it. Specifically, you can define the `insert` and `deleteMin` operations in terms of merging, and the other operations are then trivial to define, except for the list-to-heap conversion routine. That can be done easily in O(N log N) time, and with more difficulty in O(N) time. You're not required to find the most efficient solution, but it's a good exercise.

Merging leftist heaps

OK, so now we need to figure out how to merge two leftist heaps to create a new leftist heap. The basic algorithm is quite simple:

- If either heap is empty, return the other heap.
- If the first heap's minimum element is smaller than the second heap's minimum element, make a new heap (*) from the first heap's minimum element, the first heap's left subheap, and the result of merging the first heap's right subheap with the second heap.

- Otherwise, make a new heap (*) from the second heap's minimum element, the second heap's left subheap, and the result of merging the first heap with the second heap's right subheap.

(*) Here is how to make a new heap from a minimum element and two heaps: the resulting heap must have:

- the given minimum element
- a rank which is the smaller of the ranks of the two heaps, plus 1
- a left subheap which is the heap with the larger rank
- a right subheap which is the heap with the smaller rank

This algorithm will preserve the leftist heap property in the merged heap (though this fact may not be obvious).

Part A: Implementing the priority queue

In this section, you will implement the priority queue datatype as a Haskell module. All code for this section will go into the file `PriorityQueue.hs`.

The file should start with this module declaration:

```
module PriorityQueue(Pqueue,
                     empty,
                     isEmpty,
                     insert,
                     findMin,
                     deleteMin,
                     popMin,
                     fromList,
                     isValid)
where
```

The `where` in the last line is not a mistake; it's part of the syntax of module declarations and implies that the entirety of the rest of the file is part of the module.

This module declaration has an explicit list of all the functions and values that are being exported from the module. Inside the module we can (and will) be defining additional functions. Also, note that `Pqueue` (the priority queue datatype) is exported, but its constructors are not! (If you wanted to export the constructors, you would write `Pqueue(..)` there.) This makes the `Pqueue` datatype an abstract datatype as far as any client of this module is concerned; they can only interact with priority queues by passing them to the functions exported from this module (or functions which call such functions). This in turn means that the priority queue implementation can be changed without having to change any code in any other module which uses priority queues, as long as the export list is not changed. Experienced programmers recognize this as a Good Thing. In this case, priority queues will be implemented as leftist heaps, but code using this module will not be able to interact with the internals of rhe leftist heap implementation.

1. **[10]** Define the priority queue datatype, which should be called `Pqueue a`. It should be implemented as a leftist heap datatype as described previously. It should have two constructors, one called `Leaf` and the other called `Node`. See the previous discussion for what fields each

constructor should contain. Use `Int` as the datatype for ranks.

2. **[5]** Define the `empty` value, which represents an empty priority queue. It should have this type signature:

   ```
   empty :: Pqueue a
   ```

   Note that `empty` is exported from the module even though the constructors of the `Pqueue` datatype are not.

3. **[5]** Define the `isEmpty` function, which has this type signature:

   ```
   isEmpty :: Pqueue a -> Bool
   ```

   `isEmpty` returns `True` if and only if the priority queue argument is the empty priority queue.

4. **[5]** Define the `rank` function, which has this type signature:

   ```
   rank :: Pqueue a -> Int
   ```

   `rank` returns the integer rank of the priority queue argument. It should have an O(1) time complexity *i.e.* it shouldn't have to do any real computation.

5. **[30]** Define the `merge` function, which has this type signature:

   ```
   merge :: Ord a => Pqueue a -> Pqueue a -> Pqueue a
   ```

   `merge` takes two priority queues (represented as leftist heaps) and merges them, returning a new priority queue. The original priority queues are not altered. The merging algorithm was described previously. If implemented properly, this function will have O(log N) time complexity, where N is the total number of elements in both queues.

   Hint: You may find that the `@` syntax in patterns is very useful here. See the lectures or online documentation if you don't remember what `@` means in a pattern.

6. **[10]** Define the `insert` function, which has this type signature:

   ```
   insert :: Ord a => a -> Pqueue a -> Pqueue a
   ```

   `insert` "inserts" a new item into a priority queue, returning the expanded priority queue. Note that the priority queue is allowed to contain duplicated items. This function can (and should) trivially be defined in terms of the `merge` function previously defined.

7. **[5]** Define the `findMin` function, which has this type signature:

   ```
   findMin :: Ord a => Pqueue a -> Maybe a
   ```

   `findMin` finds and returns the minimum value from a priority queue. If the queue is empty, it returns `Nothing`; otherwise it returns `Just` the minimum value. This function has an O(1) time complexity and is another trivial definition.

8. **[10]** Define the `deleteMin` function, which has this type signature:

```
deleteMin :: Ord a => Pqueue a -> Maybe (Pqueue a)
```

`deleteMin` takes a priority queue, removes the smallest element, and returns a new priority queue made up of the remaining elements. If the priority queue argument is the empty queue, it returns `Nothing`, otherwise it returns `Just` the new queue. Again, this can (and should) be trivially defined in terms of the `merge` function.

9. **[5]** Define the `popMin` function, which has this type signature:

   ```
   popMin :: Ord a => Pqueue a -> Maybe (a, Pqueue a)
   ```

   `popMin` takes a priority queue, removes the smallest element, and returns both the removed element and a new priority queue made up of the remaining elements. Again, if the priority queue argument is the empty queue, it returns `Nothing`, otherwise it returns `Just` the (removed item, new queue) tuple.

10. **[20]** Define the `fromList` function, which has this type signature:

    ```
    fromList :: Ord a => [a] -> Pqueue a
    ```

    `fromList` takes a list of items of an orderable type `a` and converts them into a priority queue. *Hint:* This function can be defined in one line (actually in three words!) without using recursion by using a higher-order function which we've studied in class. However, you aren't required to define it this way.

11. **[30]** Define the `isValid` function, which has this type signature:

    ```
    isValid :: Ord a => Pqueue a -> Bool
    ```

    The purpose of this function is to test whether a priority queue value is really a leftist heap. The datatype we are using for leftist heaps is not powerful enough to enforce the leftist heap invariant, so we can use this function to check that the invariant is in fact maintained. Of course, just returning `True` will get you no credit.

    In order to validate a leftist heap, you must establish the following:

    1. A non-leaf node's rank is a positive integer.
    2. A non-leaf node's left subheap has a rank which is at least as large as that of its right subheap.
    3. A non-leaf node's rank is one larger than the rank of its right subheap.
    4. A non-leaf node's data item is less than or equal to the minimum item in either subheap.
    5. The subheaps are also valid leftist heaps.

    Note that leaves are automatically valid leftist heaps.

    You may use the `rank` and `findMin` functions you defined above in your solution to this problem.

These are all the functions you need to define in this module.

Part B: Implementing the heapsort algorithm

In this section, you will use the priority queue module from the last section to implement a sorting algorithm known as "heapsort". All code for this section will go into the file `HeapSort.hs`.

The heapsort algorithm

Priority queues implemented using heaps naturally lend themselves to implementing a sorting algorithm, which is not surprisingly referred to as "heapsort". The idea is very simple: you convert a list of values into a priority queue (implemented as a heap), then you repeatedly extract the minimum value from the heap, storing the minimum values as successive values in a list, until there are no more values.

One pitfall to avoid is that it's better to store the minimum values in reverse order in a list and then reverse the entire list at the end of the sort. Adding values to the end of a list is very inefficient and can ruin the asymptotic time complexity of the algorithm, which should be O(N log N). This is easy to compute: converting a list of length N into a heap requires (for a naive algorithm) O(N log N) steps because adding each value to the heap is O(log N). Removing the minimum element from a heap and reconstituting the rest of the elements as a new heap requires O(log N) steps per removal, and N items need to be removed, giving O(N log N) for that part. Finally, reversing the list is O(N). Therefore, the total time complexity is O(N log N).

Writing the code

The first few lines of your module should be the following:

```
module HeapSort where

import qualified PriorityQueue as Q
```

The first line declares the module `HeapSort`, which will export all of its definitions (in other words, there is no export list). The second line imports the `PriorityQueue` module which you wrote in the previous section. Here we are doing a "qualified import" and referring to the module as `Q`. This means that any function or value from the `PriorityQueue` module must have the prefix `Q.`. For instance, the `isEmpty` function from the `PriorityQueue` module would be referred to in this module as `Q.isEmpty`.

For this module, you only have to write a single function.

1. **[20]** Define the `sort` function, which has this type signature:

   ```
   sort :: Ord a => [a] -> [a]
   ```

   This function should implement the heapsort algorithm described above. The implementation can be very short (less than ten lines). *Hint:* Use the `popMin` function from the `PriorityQueue` module.