Quickcheck: Instructions

Help

When you're ready to submit your solution, go to the assignments list.

Attention: You are allowed to submit **a maximum of 5 times**! for grade purposes. Once you have submitted your solution, you should see your grade and a feedback about your code on the Coursera website within 20 minutes. If you want to improve your grade, just submit an improved solution. The best of all your first 5 submissions will count as the final grade. You can still submit after the 5th time to get feedback on your improved solutions, however, these are for research purposes only, and will not be counted towards your final grade.

Download the quickcheck.zip handout archive file and extract it somewhere on your machine.

In this assignment, you will work with the ScalaCheck library for automated specification-based testing.

You're given several implementations of a purely functional data structure: a heap, which is a priority queue supporting operations `insert`, `meld`, `findMin`, `deleteMin`. Here is the interface:

```
trait Heap {
  type H // type of a heap
  type A // type of an element
  def ord: Ordering[A] // ordering on elements

  def empty: H // the empty heap
  def isEmpty(h: H): Boolean // whether the given heap h is empty

  def insert(x: A, h: H): H // the heap resulting from inserting x into h
  def meld(h1: H, h2: H): H // the heap resulting from merging h1 and h2

  def findMin(h: H): A // a minimum of the heap h
  def deleteMin(h: H): H // a heap resulting from deleting a minimum of h
}
```

All these operations are *pure*; they never modify the given heaps, and may return new heaps. This purely functional interface is taken from Brodal & Okasaki's paper, *Optimal Purely Functional Priority Queues*.

Here is what you need to know about priority queues to complete the assignment. In a nutshell, a priority queue is like a queue, except it's not first-in first-out, but whatever-in minimum-out. Starting with the `empty` queue, you can construct a new non-empty bigger queue by recursively `insert`ing an element. You can also `meld` two queues, which results in a new queue that contains all the elements of the first queue and all the elements of the second queue. You can test whether a queue is empty or not with `isEmpty`. If you have a non-empty queue, you can find its minimum with `findMin`. You can also get a smaller queue from a non-empty queue by deleting the minimum element with `deleteMin`.

Only one of the several implementations you are given is correct. The other ones have bugs. Your goal is to write some properties that will be automatically checked. All the properties you write should be satisfiable by the correct implementation, while at least one of them should fail in each incorrect

implementation, thus revealing it's buggy.

You should write your properties in the body of the `QuickCheckHeap` class in the file `src/main/scala/quickcheck/QuickCheck.scala`. In this class, the heap operates on `Int` elements with the natural ordering, so `findMin` finds the least integer in the heap.

As an example of what you should do, here is a property that ensures that if you insert an element into an empty heap, then find the minimum of the resulting heap, you get the element back:

```
property("min1") = forAll { a: Int =>
  val h = insert(a, empty)
  findMin(h) == a
}
```

We also recommend you write a *generator* of heaps, of abstract type `H`, so that you can write properties on any random heap, generated by your procedure. For example,

```
property("gen1") = forAll { (h: H) =>
  val m = if (isEmpty(h)) 0 else findMin(h)
  findMin(insert(m, h))==m
}
```

To get you in shape, here is an example of a generator for maps of type `Map[Int,Int]`.

```
lazy val genMap: Gen[Map[Int,Int]] = for {
  k <- arbitrary[Int]
  v <- arbitrary[Int]
  m <- oneOf(value(Map.empty[Int,Int]), genMap)
} yield m.updated(k, v)
```

In order to get full credit, all tests should pass, that is you should correctly identify each buggy implementation while only writing properties that are true of heaps. You are free to write as many or as few properties as you want in order to achieve a full passing suite.

Hints

Here are some possible properties we suggest you write.

- If you insert any two elements into an empty heap, finding the minimum of the resulting heap should get the smallest of the two elements back.
- If you insert an element into an empty heap, then delete the minimum, the resulting heap should be empty.
- Given any heap, you should get a sorted sequence of elements when continually finding and deleting minima. (Hint: recursion and helper functions are your friends.)
- Finding a minimum of the melding of any two heaps should return a minimum of one or the other.