coursera.org/learn/progfun2/programming/DF4y7/quickcheck

Note: If you have paid for the Certificate, please make sure you are submitting to the required assessment and not the optional assessment. If you mistakenly use the token from the wrong assignment, your grades will not appear

Attention: You are allowed to submit **an unlimited number of times**. 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 submissions will count as the 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 A

  def ord: Ordering[A]

  def empty: H

  def isEmpty(h: H): Boolean

  def insert(x: A, h: H): H

  def meld(h1: H, h2: H): H

  def findMin(h: H): A

  def deleteMin(h: H): 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*.

A priority queue is a queue, in which each element is assigned a "priority". In classical queues, elements can be retrieved in first-in, first-out order, whereas in a priority queue, elements are retrieved as per the priority they are assigned. As such, classical queues are therefore just priority queues where the priority is the order in which elements are inserted.

As seen in the above interface, we can create a queue by

- instantiating an *empty* queue.
- inserting an element into a queue (with an attached priority), thereby creating a new queue.
- melding 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.

In addition, we can 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. In this assignment, the heap operates on Int elements with their values as priorities, so findMin finds the least integer in the heap.

You are given multiple implementations of IntHeaps in file src/main/scala/quickcheck/Heap.scala. Only one of these is correct, while 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.

Part 1: A Heap Generator

Before checking properties, we must first generate some heaps. Your first task is to implement such a generator:

```
lazy val genHeap: Gen[H] = ???
```

For doing this, you can take inspiration from the lecture on generators and monads. Here are some basic generators that you can combine together to create larger ones:

- arbitrary[T] is a generator that generates an arbitrary value of type T. As we are interested in IntHeaps it will generate arbitrary integer values, uniformly at random.
- oneOf(gen1, gen2) is a generator that picks one of gen1 or gen2, uniformly at random.
- const(v) is a generator that will always return the value v.

You can find many more useful ones either in the ScalaCheck user guide or in the Scaladocs.

For instance, we can write a generator for maps of type Map[Int, Int] as follows:

```
lazy val genMap: Gen[Map[Int,Int]] = oneOf(

  const(Map.empty[Int,Int]),

  for {

    k <- arbitrary[Int]

    v <- arbitrary[Int]

    m <- oneOf(const(Map.empty[Int,Int]), genMap)

  } yield m.updated(k, v)

)
```

Part 2: Writing Properties

Now that you have a generator, you can write property-based tests. The idea behind property-based testing is to verify that certain properties hold on your implementations. Instead of specifying exactly which inputs our properties should satisfy, we instead generate random inputs, and run each property test on these randomly generated inputs. This way we increase the likelihood that our implementation is correct.

For example, we would like to check that adding a single element to an empty heap, and then removing this element, should yield the element in question. We would write this requirement as follows:

```
property("min1") = forAll { a: Int =>

  val h = insert(a, empty)

  findMin(h) == a

}
```

Another property we might be interested in is that, for any heap, adding the minimal element, and then finding it, should return the element in question:

```
property("gen1") = forAll { (h: H) =>

  val m = if (isEmpty(h)) 0 else findMin(h)

  findMin(insert(m, h)) == m

}
```

In src/main/scala/quickcheck/QuickCheck.scala, write some more properties that should be satisfied. Your properties should at least cover the following relevant facts:

- 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.

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. Your properties should cover all of the above-stated relevant facts. You are free to write as many or as few properties as you want in order to achieve a full passing suite.

Note that this assignment asks you to write tests whose content captures all of the above relevant facts, and whose execution correctly differentiates correct from incorrect heaps among the heaps given to you. You need not worry about additional buggy heaps that someone else might write.