Problem Sheet 4

Question 1:

Here is the design I have come up with. There are a lot of improvements that can be made, but for the static problem given here, it should work.

Here is a `Shape` trait that I have made:

```scala
trait Shape {
  def getSize() : Double;
  def isShape(a : Shape) : Boolean;
  }
```

And here are the four classes for each shape:

```scala
class Circle(var radius : Double) extends Shape {

  override def getSize(): Double = {
    PI * radius * radius
  }

  override def isShape(a: Shape): Boolean = a match {
    case _: Circle => true
    case _: Ellipse => true
    case _ => false
  }

  def setRadius(newRadius : Double) : Unit = {
    this.radius = newRadius
  }
}
```

```scala
class Ellipse(var semi_major : Double, var semi_minor : Double) extends Shape {

  override def getSize(): Double = {
    PI * semi_major * semi_minor
  }

  override def isShape(a: Shape): Boolean = a match {
    case _: Ellipse => true
    case _ => false
  }

  def setSemiMajor(newSemiMajor : Double) = {
    this.semi_major = newSemiMajor
  }

  def setSemiMinor(newSemiMinor: Double) = {
    this.semi_minor = newSemiMinor
  }

}
```

```
class Rectangle(var length: Double, var width: Double) extends Shape {

  override def getSize(): Double = {
    length * width
  }

  override def isShape(a: Shape): Boolean = a match {
    case _: Rectangle => true
    case _ => false
  }

  def setLength(newLength : Double): Unit = {
    this.length = newLength
  }

  def setWidth(newWidth : Double) : Unit = {
    this.width = newWidth
  }
}
```

```
class Square(var length : Double) extends Shape {

  override def getSize(): Double = {
    length * length
  }

  override def isShape(a: Shape): Boolean = a match {
    case _: Square => true
    case _: Rectangle => true
    case _ => false
  }

  def setLength(newLength : Double) : Unit = {
    this.length = newLength;
  }
}
```

Now, this design doesn't have traits that classify any of these shapes. A better design would also have involved the separate traits `Smooth` and `Pointy` to be implemented by Circle and Ellipse for Smooth, and by Rectangle and Square for Pointy.

Having loosely coupled class references by the use of traits make it useful for code that grows as time progresses. If the use of Traits was not implemented here or not thought of being used, then the program would get hard to manage as it increased in size and more shape classes were added to the program.

It does make the initial programming of the program a bit more tiresome, but it pays out as the program grows and evolves. I used my knowledge of Interface Segregation from SOLID principles in order to minimise the size of the Trait, so that as the program expands, the interface need not be touched, or manipulated in such a way to cause more problems refactoring than developing.

Question 2:

The following code will end up printing `Accepted for rendering`, and then the next line, print `Accepted for ray-trace rendering`.

From the behaviour of the program, it looks as if the definition of the variable and its class methods and attributes take precedence. Here's a possible way of changing the output of `RayTraceRenderer`. At runtime, Scala selects the parent class's implementation. To fix this, just introduce the keyword `override`.

```
class RayTracingRenderer extends Renderer {
  override def accept(a: Triangle) : Unit = a match {
    case opaque: OpaqueTriangle => println("Accepted for ray-trace rendering.")
    case _ => println("Accepted for rendering")
  }
}
```

Though, the signature of the method needed to be changed slightly, as well as the logic too.

Question 3

Reading through the class implementation of `HashSet`, we find the following datatype invariant:

```
/* The Hashset class holds the following invariant:
 * - For each i between  0 and table.length, the bucket at table(i) only contains
elements whose hash-index is i. * - Every bucket is sorted in ascendent hash
order * - The sum of the lengths of all buckets is equal to contentSize. */
```

```
class HashSetTest extends AnyFunSuite {
  val hashSet = new mutable.HashSet[Int];

  test("Testing functionality of add") {
    hashSet.clear()
    hashSet.add(1) ; hashSet.add(2) ; hashSet.add(3)
    assert(hashSet.size == 3)
  }

  test("Testing the functionality of remove") {
    hashSet.clear() ; hashSet.add(5) ; hashSet.remove(5)
    assert(!hashSet.contains(5))
  }

  test("Testing the functionality of contains") {
    hashSet.clear()
    hashSet.add(1); hashSet.add(2)
    assert(hashSet.contains(2))
  }

  test("Testing the functionality of contains 2") {
    hashSet.clear()
    hashSet.add(14)
    assert(!hashSet.contains(1))
  }

  test("Testing the functionality of size") {
    hashSet.clear()
    hashSet.add(1) ; hashSet.add(2)
    assert(hashSet.size != 1)
  }

  test("Testing the functionality of isEmpty") {
    hashSet.clear()
    assert(hashSet.isEmpty)
  }

  test("Testing removing an element from an empty list") {
    hashSet.clear()
    assertThrows[NoSuchElementException](hashSet.remove(5))
(reflect.ClassTag[NoSuchElementException])  // I can't seem to get this test to
work

  }

}
```

Question 4

Here is the abstract specification of the `Stack[A]` interface:

```
trait Stack[A] {
  /** Pushes an element onto the stack.
    *
    * @param x The element to push onto the stack.
    * post : S = [A] ++ S
    */
  def push(x: A): Unit

  /** Pops the most recently added element from the stack.
    *
    * @return The most recently added element.
    * @throws NoSuchElementException if the stack is empty.
    * post : (A:Ss) = Ss
    */
  def pop(): A

  /** Tests whether the stack is empty.
    *
    * @return true if the stack is empty, false otherwise.
    */
  def isEmpty: Boolean
}
```

Question 5

a) We will have to add another conditional logic statement in the postcondition for the specification of `add`. It will look like the following:

```
/** Add elem to the set.
* post: S = S_0 ∪ {elem | elem ∈ {0..N-1}} */
def add(elem: Int): Unit
```

b) We will need to change most of the post conditions set for most of the method specifications:

```
/** state: S : P Int
* init: S = {x = 0 | x ∈ [0..N)} */
class BitMapSet(size : Int) {

        private val seq = new Array[Boolean](size)

        /** Add elem to the set.
        * post: S = {x_i = 1 | i = elem} */
        def add(elem: Int): Unit = {
                require(0 <= elem && elem < size)
                this.seq(elem) == true
        }
        /** Does the set contain elem?
        * post: S = x_i (if x_i is zero, then its false, otherwise true)*/
        def contains(elem: Int): Boolean = {
                if (elem >= size || elem < 0) false
                else this.seq(elem)
        }

        /** Remove elem from the set.
        * post: S => x_i = 0 (if the i-th element of the map is 1, make it
zero)*/
        def remove(elem: Int): Unit = {
                require(0 <= elem && elem < size)
                this.seq(elem) = false
        }

        /** The size of the set.
        * post: S = S0 ∧ returns |{n | x_n == 1}| */
        def size: Int = {
                var total = 0
                for (i <- 0 until size) {
                        if (this.seq(i)) total += 1
                }
                return total
        }
}
```

The definitions have been changed for them to work logically with the specifications.

Question 6

a) The description of the operation is brief, but clear. It is missing the pre and post conditions of such an operation.

b) Here is such a defined specification for the `head` attribute of the set:

```
/** Represents a set of integers with no duplicate elements.
  *
  * state: S : P Int - A set of integers.
  * init: S = {} - Initially, the set is empty.
  */
trait IntSet {

  /** Returns an element from the set.
    *
    * @return An integer element from the set.
    * @throws NoSuchElementException if the set is empty.
    *
    * postcondition: The returned value is an element of S.
    *                 S remains unchanged.
    */
  def head: Int

        ...
}
```

c) Here is the implementation of `head` in the `BitMapSet`

```
/** Returns the first element of the set
        * post: S = S0 ∧ returns |this.seq(0)| */
        def head : A = {
                var found = false ; i = 0
                while (i < n && found == false) {
                        if (this.seq(i)) found == true
                        i += 1
                }
                return (i-1 => A)
        }
```

Question 7

a) Here is the augmented trait operation with the specific pre and post conditions for `delete`:

```
/** Delete the number stored against name (if it exists).
    * post: If name was in dom book then book = book0 \ {name} ∧ returns true,
    *       otherwise book = book0 ∧ returns false
    */
  def delete(name: String): Boolean
```

b) To implement the `delete` operation in the phone book represented by the adjusted implementation, the following needs to be done. First, the pre and post conditions need to account for the fact that they are disjoint lists, so as to not leave 'holes' in the arrays.

```
class ArraysBook {
  var names: Array[String] = ???
  var numbers: Array[String] = ??? // not sure how these should be initialised

  // Other methods like store and recall

  /** Delete the number stored against name (if it exists).
    * @param name The name of the entry to delete.
    * @return true if the entry was found and deleted, false otherwise.
    */
  def delete(name: String): Boolean = {
    val index = names.indexOf(name)
    if (index != -1) {
      // Remove element by shifting the subsequent elements left
      names = names.take(index) ++ names.drop(index + 1)
      numbers = numbers.take(index) ++ numbers.drop(index + 1)
      true
    } else {
      false
    }
  }
}
```

Question 8

Here are the function implementations I'd use in order to have an $O(\log n)$ runtime. Unfortunately, this does make some of the other functions less optimised.

```
// Linear runtime for finding the correct place to insert
def store(name: String, number: String): Unit = {
    val index = names.search(name).insertionPoint
    names = (names.take(index) :+ name) ++ names.drop(index)
    numbers = (numbers.take(index) :+ number) ++ numbers.drop(index)
  }
```

```
def recall(name: String): String = {
    val index = names.search(name).searchResult
    if (index >= 0) numbers(index) else throw new NoSuchElementException(s"$name
not found")
  }
```

```
def delete(name: String): Boolean = {
    val index = names.search(name).searchResult
    if (index >= 0) {
      names = names.take(index) ++ names.drop(index + 1)
      numbers = numbers.take(index) ++ numbers.drop(index + 1)
      true
    } else false
  }
```

```
def isInBook(name: String): Boolean = names.search(name).searchResult >= 0
```

```
private def search(arr: Array[String], key: String): Int = {
    var low = 0
    var high = arr.length - 1
    while (low <= high) {
      val mid = (low + high) >>> 1
      val midVal = arr(mid)

      if (midVal < key) low = mid + 1
      else if (midVal > key) high = mid - 1
      else return mid // key found
    }
    -1 // key not found
  }
```

Question 9

For such a specification of this Bag datatype, here we will need to specify two functions required for this trait. One is to add an element to the bag, and the other would be retrieve the count of the bag. Here is the abstract specification of the bag:

```
trait Bag[A] {
  /** Adds an integer to the bag.
    *
    * @param x The integer to add. It must be in the range [0..MAX).
    * @return Unit
    * post: If x is within the range [0..MAX), then bag(x) is increased by 1.
    *       Otherwise, the state of bag remains unchanged.
    */
  def add(x: Int): Unit

  /** Finds the number of occurrences of an integer in the bag.
    *
    * @param x The integer to query. It can be any integer, but counts for
integers outside [0..MAX) are always 0.
    * @return The number of occurrences of x in the bag.
    * post: Returns the current count of x in the bag, which is >= 0. For x
outside [0..MAX), it returns 0.
    */
  def count(x: Int): Int
}
```

And here is the implementation of such a bag where in `val c = new Array[Int](MAX)`, the elements at each index record the number of copies of the index in the bag.

```
class ArrayBag(MAX: Int)[A] extends Bag {
  // Initialize the array to store counts. All counts are initially 0.
  private val c = new Array[Int](MAX)

  /** Adds an integer to the bag.
    *
    * @param x The integer to add. It must be in the range [0..MAX).
    * @return Unit
    */
  override def add(x: Int): Unit = {
    if (x >= 0 && x < MAX) {
      c(x) += 1
    }
  }

  /** Finds the number of occurrences of an integer in the bag.
    *
    * @param x The integer to query.
    * @return The number of occurrences of x in the bag.
    */
  override def count(x: Int): Int = {
    if (x >= 0 && x < MAX) c(x) else 0
  }
}
```

(P.S I was unclear on how to add an appropriate abstract function).

Question 10

This approach will use the logic that the Counting Sort algorithm uses. Here is how it would be done with the use of the `Bag` specification

```
def sort(unsorted : Array[Int]) : Array[Int] = {
        // turning the list into a bag:
        val n = unsorted.size ; val sorted = new Array[Int](n) ; val mid = new
Bag[Int]
        for (i <- 0 until n) {
                mid.add(unsorted(i))
        }
        var i = 0; var k = 0
        while (i < mid.MAX) {
                for (j <- 0 until mid(i)) {
                        sorted(k) = mid(i)
                        k += 1
                }
                i += 1
        }
        sorted
}
```