Tuan Nguen

Question 1:

```
object Question1{
  def main(args : Array[String]) = {
    val mySet = new scala.collection.mutable.HashSet[String]

    // Test add and contains method
    mySet.add("Dog"); mySet.add("Cat"); mySet.add("Elephant");
mySet.add("Milk")
    assert(mySet.contains("Dog") == true)
    assert(mySet.contains("Cat") == true)
    assert(mySet.contains("Elephant") == true)
    assert(mySet.contains("Milk") == true)

    // Test remove and contains method
    mySet.remove("Cat")
    assert(mySet.contains("Dog") == true)
    assert(mySet.contains("Cat") == false)
    assert(mySet.contains("Elephant") == true)
    assert(mySet.contains("Milk") == true)

    // Test size method
    assert(mySet.size == 3)
    mySet.add("Cat")
    assert(mySet.size == 4)

    // Test isEmpty method
    assert(mySet.isEmpty == false)
    mySet.remove("Dog"); mySet.remove("Cat");
mySet.remove("Elephant"); mySet.remove("Milk")
    assert(mySet.isEmpty == true)

    mySet.add("Dog"); mySet.add("Dog")
    assert(mySet.contains("Dog") == true)
    println(mySet.add("Dog"))
    println(mySet.remove("Cat"))
    mySet.add(1)
  }
}
```

If we try to add an element, which is already in the set, the function will return false. Otherwise, it will return true. Analogously for the remove function – if the element that we are trying to remove is present in the set, the function will return true, false otherwise. The last line will throw an error because we are trying to add an integer to the set, which consists only of strings.

Question 2:

```
/** state: Q: P A(power set of datatype A)
  * init: Q = {}
trait Stack[A]{
  /** Add element to the top of the stack
    * post: Q = Q0 ++ {elem}
  def push(elem : A)

  /** Remove the element at the top and return it
    * post: Q = init Q0 && return (last Q0)
  def pop()

  /** Returns true if the stack is empty
    * post: Q == {}
  def isEmpty() : Boolean
}
```

Question 3:

a)

```
/** state: S: P(0..N-1)
  * init: S = {} */
trait IntSet{
  /** Add elem to the set.
    * Pre:
    * post: S = S0 U {elem} && 0 <= elem < N */
  def add(elem : Int)

  /** Does the set contain elem?
    * post: S = S0 && returns elem && 0 <= elem < N */
  def isIn(elem : Int) : Boolean

  /** Remove elem from the set.
    * post: S = S0 - {elem} && 0 <= elem < N */
  def remove(elem : Int)

  /** The size of the set.
    * post: S = S0 && returns #S */
  def size() : Int
}
```

b)

```
class BitMapSet extends IntSet{
  private val MAX = 1000 // max elements in the set
  private var set = new Array[Boolean](MAX)
  // Abs: BitMapSet = set[0..1000)
```

```
  // DTI: set.size = 1000 && set(i) = true or false for every i in
[0..1000)

  /** Add elem to the set */
  def add(elem : Int) = {
    assert(elem < 1000)
    set(elem) = true
  }

  /** Returns whether the elem is in set */
  def isIn(elem : Int) : Boolean = {
    assert(elem < 1000)
    set(elem)
  }

  /** Remove the element form the set */
  def remove(elem : Int) = {
    assert(elem < 1000)
    set(elem) = false
  }

  /** Returns the size of the set */
  var count = 0; var i = 0
  def size() : Int = {
    while(i < 1000){
      if(set(i)){ count += 1 }
      i += 1
    }
    count
  }
}
```

Question 4:

a) The description of the function is not the exact specification. For example if I return the first element and delete the rest. That would meet the description but it is not the desired result.

b) A suitable specification would be:

```
/** Returns the first element of the set
  * Pre: the set is not empty
  * Post: set = set0 && returns (head set) */
```

c)

```
class BitMapSet extends IntSet{
  private val MAX = 1000 // max elements in the set
  private val set = new Array[Boolean](MAX)
  // Abs: BitMapSet = set[0..1000)
```

```
  // DTI: set.size = 1000 && set(i) = true or false for every i in
[0..1000)

  /** Add elem to the set */
  def add(elem : Int) = {
    assert(elem < 1000)
    set(elem) = true
  }

  /** Returns whether the elem is in set */
  def isIn(elem : Int) : Boolean = {
    assert(elem < 1000)
    set(elem)
  }

  /** Remove the element form the set */
  def remove(elem : Int) = {
    assert(elem < 1000)
    set(elem) = false
  }

  /** Returns the size of the set */
  var count = 0; var i = 0
  def size() : Int = {
    while(i < 1000){
      if(set(i)){ count += 1 }
      i += 1
    }
    count
  }

  /** Returns the head of the set */
  def head() : Int = {
    assert(!set.forall(x => x == false)) // Test if the set is non-
empty
    var i = 0
    while(!set(i)){ i += 1 }
    i
  }
}
```

Question 5:

```
// Easch implementation of this trait represents a mapping from names
// (Strings) to numbers (also Strings).
trait Book{
  // State: book : String -|-> String
```

```
  // Init:  book = {}

  // Add the maplet name -> number to the mapping
  // Post: book = book_0 (+) {name -> number}
  def store(name:String, number:String)

  // Return the number stored against name
  // Pre: name in dom book
  // Post: book = book_0 && returns book(name)
  def recall(name:String) : String

  // Is name in the book?
  // Post: book = book_0 && returns name in dom book
  def isInBook(name: String): Boolean

  /** Delete the number stored against name (if it exists)
    * Post: if name is in book: book = book0 - {name} && return true
    *       if not in book    : book = book0 && return false
  def delete(name : String) : Boolean
}
```

b)

```
/** Delete the number(if it exists) */
def delete(name : String) : Boolean = {
  val index = find(name)
  if(index != count){
    names(index) = names(count-1)
    count -= 1
    true
  }
  else false
}
```

Question 6:

```
class ArraysBook extends Book{
  private val MAX = 1000 // max number of names we can store
  private val names = new Array[String](MAX)
  private val numbers = new Array[String](MAX)
  private var count = 0
  /** These variables together represent the mapping
    * Abs: book =
    * { names(i) -> numbers(i) | i <- [0..count) }
    * DTI: 0 <= count <= MAX &&
    * entries in names[0..count) are distinct */

  /** Return an index i and a Boolean s.t.
```

```
  * names(i) >= name > names(i-1) and the
  * Boolean tells whether such name is found */
def find(name: String) : (Int, Boolean) = {
  // Invariant: name is in name[i..j) && 0 <= i <= j <= count
  // Variant: j - i
  var i = 0; var j = count
  while(i < j){
    val med = (i + j) / 2
    if(names(med) < name) i = med + 1 else j = med
  }
  // i = j, so name[0..i) < name <= name[i..count)
  if(name == names(i)) (i, true) else (i, false)
}

/** Return the number stored against name */
def recall(name: String) : String = {
  val i = find(name)._1
  assert(i < count)
  numbers(i)
}

/** Is name in the book? */
def isInBook(name: String) : Boolean = {
  find(name)._2 == true
}

/** Add the maplet name -> number to the mapping */
def store(name: String, number: String) = {
  val (i, bool) = find(name)
  if(!bool){
    assert(count < MAX);
    // Move all the names after i one place to the right
    // Invariant: names[0..i) = names0[0..i) < name &&
    //            names[j+1..count+1) = names0(j..count)
    //            names[j+1..count+1) > name &&
    //            i <= j <= count
    var j = count
    while(j >= i){ names(j+1) = names(j); j -= 1 }
    names(i) = name; count += 1
  }
  numbers(i) = number
}

/** Delete name from the book */
def delete(name : String) : Boolean = {
  val (i, bool) = find(name)
```

```
    if(bool){
      // Move all the elements after i one place to the left
      // Invariant: names[0..i) = names0[0..i) < name &&
      //            names[i..j) = names[i+1..j+1) &&
      //            names[i..j) > name
      //            i + 1 <= j < count
      var j = i
      while(j < count - 1){ names(j) = names(j+1); j += 1 }
      count -= 1
    }
    bool
  }
}
```

The find function finds a name in *O*(logn) time because it uses binary search. Both the `store` and the `delete` function take *O*(n) time because it shifts the elements one place to the right and left respectively. The shifting cannot be avoided because the phonebook has to maintain the property that it is sorted.

Question 7:

```
trait Bag{
  /** State: bag: Int -|-> Int
    * Init: bag = {} */

  /** Add a number to the bag
    * Post: bag(j) = bag0(j) for j != i
    *       bag(i) = bag0(i) + 1 */
  def add(number : Int)

  /** Find the number of copies of an integer
    * Pre: the number is in range
    * Post: bag = bag0 && returns bag(number) */
  def copies(number : Int) : Int
}

class ArrayBag extends Bag{
  private val MAX = 1000
  private val c = new Array[Int](MAX)
  /** These variables together represent the bag
    * Abs: bag = {c(i) copies of i | i <- [0..MAX)} */

  /** Add a number to the bag */
  def add(number : Int) = {
    c(number) += 1
  }
```

```
  /** Find a number of copies of an integer */
  def copies(number : Int) : Int = {
    assert(number < MAX)
    c(number)
  }
}
```

Question 8:

```
trait Bag{
  /** State: bag: Int -|-> Int
    * Init: bag = {} */

  /** Add a number to the bag
    * Post: bag(j) = bag0(j) for j != i
    *       bag(i) = bag0(i) + 1 */
  def add(number : Int)

  /** Find the number of copies of an integer
    * Pre: the number is in range
    * Post: bag = bag0 && returns bag(number) */
  def copies(number : Int) : Int
}

class ArrayBag extends Bag{
  private val MAX = 1000
  private val c = new Array[Int](MAX)
  /** These variables together represent the bag
    * Abs: bag = {c(i) | i <- [0..MAX)} */

  /** Add a number to the bag */
  def add(number : Int) = {
    c(number) += 1
  }

  /** Find a number of copies of an integer */
  def copies(number : Int) : Int = {
    assert(number < MAX)
    c(number)
  }

  /** Prints the number in a bag */
  def printBag() = {
    // Invariant: all copies of c[0..i) are printed
    //         && 0 <= i <= 1000
    var i = 0
    while(i < 1000){
```

```
      for(j <- 1 to c(i)){ print(i + " ") }
      i += 1
    }
  }
}

object Question8{
  def main(args : Array[String]) = {
    val bag = new ArrayBag
    for(i <- 0 until args.size){ bag.add(args(i).toInt) }
    bag.printBag()
  }
}
```