Quiz Section 3: ADTs – Solutions

The next few problems concern the following ADT:

```
/**
 * Represents an immutable collection of integers.
 *
 * Clients can think of a set as a list of integers. However, they can only ask
 * if an integer is present or not. The order of the integers and the number of
 * times an integer appears in the list are inaccessible and do not matter.
 */
public interface IntSet {
  /**
   * Determines whether n is in the list.
   * @param n the number to look for in the list
   * @returns contains(n, obj), where
   *     contains(n, nil)    := false
   *     contains(n, m :: L) := true           if m = n
   *     contains(n, m :: L) := contains(n, L) if m /= n
   */
  public boolean contains(int n);

  /**
   * Creates and returns a new list containing n as well as all of obj.
   * @param n the number to add to the new list.
   * @returns n :: obj
   */
  public IntSet add(int n);

  /** ... */
  public IntSet remove(int n);
}
```

Task 1 – Teacher's Set [12 pts]

Answer the following questions about the specification of `IntSet`.

a) Explain in your own words what `@return n :: obj` means. In particular, what is "`obj`" in this context? Why does this mathematical expression make sense?

> This means creating the list we return by cons-ing n onto the front of the list that `obj` represents. Since the abstract state of `obj` is also a list, this is a valid operation.

b) Suppose that we have an `IntSet T` whose abstract state is the list $1 :: 2 :: 3 ::$ nil. What mathematical value is returned by the expression `T.add(4)` according to the `add` function specification.

> That returns $4 :: 1 :: 2 :: 3 ::$ nil.

c) Write a specification for the method `remove`. It should return a list that contains all of the numbers in the current list *except* for the number `n`, which should no longer be present.
(Hint: your spec should include a math definition similar to that of `contains`)

```
/**
 * Creates and returns a new list containing all of the numbers of obj
 * except for the number n, which is no longer included.
 * @param n the number to not include in the new list.
 * @returns remove(n, obj), where
 *      remove(n, nil)    := nil
 *      remove(n, m :: L) := remove(n, L)      if m = n
 *      remove(n, m :: L) := m :: remove(n, L) if m /= n
 */
public IntSet remove(int n);
```

Task 2 – Jumping Through Dupes [12 pts]

In this problem, we will return to the original specification of `IntSet`, whose abstract state is a list of elements possibly containing duplicates. We will consider three different concrete representations for it:

```
    public class IntSetImpl implements IntSet {

(1)   // AF: obj = this.elems
      private int[] elems;

(2)   // AF: obj = this.elems
      // RI: this.elems contains no dups
      private int[] elems;

(3)   // AF: obj = this.elems
      // RI: this.elems is sorted
      private int[] elems;

      public IntSetImpl(int[] elems) {
        this.elems = elems;
      }
```

For each of the methods shown below, state the concrete representations (1–3) for which it would satisfy the specification of the method in `IntSet`. In each case, briefly explain why.

a)

```
public boolean contains(int n) {
  return Arrays.binarySearch(this.elems, n) >= 0;
}
```

Note: Binary Search is an algorithm that finds the position of a target value within a sorted array.

This implementation satisfies the specification only with concrete representation (3). When the array is not sorted, `binarySearch` is not guaranteed to find the element when present.

b)

```
public boolean contains(int n) {
  for (int i = 0; i < this.elems.length; i++) {
    if (this.elems[i] == n)
      return true;
  }
  return false;
}
```

This implementation satisfies the specification with any of the concrete representations because it does not require any representation invariant to hold.

c)

```
public IntSet add(int n) {
  if (this.contains(n)) {
    return this;
  } else {
    int[] newElems = new int[this.elems.length + 1];
    System.arraycopy(this.elems, 0, newElems, 1, this.elems.length);
    newElems[0] = n;
    return new IntSetImpl(newElems);
  }
}
```

For System.arraycopy() syntax, see: JavaDoc

This satisfies the specification of add with concrete representations (1–2). This holds trivially for (1) since it has no representation invariant, and it holds with (2) because this implementation ensures no duplicates. It would not satisfy the spec with concrete representation (3) because it does not ensure that the array is sorted.

d)

```
public IntSet remove(int n) {
  for (int i = 0; i < this.elems.length; i++) {
    if (this.elems[i] == n) {
      int[] newElems = new int[this.elems.length - 1];
      System.arraycopy(this.elems, 0, newElems, 0, i);
      System.arraycopy(this.elems, i+1, newElems, i,
          this.elems.length - i - 1);
      return new IntSetImpl(newElems);
    }
  }
  return this;
}
```

This satisfies the specification with concrete representations (2). It works with (2) because removing an element preserves the fact that there are no duplicates. It also preserves the sorting property required by (3); however, it still does not work with (3) or (1) because removing a single element does not leave an array not containing the element if there was more than one copy in the array.

Task 3 – Hold Down the Sort [12 pts]

Consider the following implementation of `IntSetImpl`, which ensures that the representation invariant is satisfied by sorting the elements in the constructor:

```
public class IntSetImpl implements IntSet {
    // AF: obj = this.elems
    // RI: this.elems is sorted in ascending order
    private int[] elems;

    public IntSetImpl(int[] elems) {
        this.elems = elems;

        // Put the elements in sorted order.
        for (int i = 1; i < elems.length; i++) {
            int key = elems[i];
            int j = i - 1;
            while (j >= 0 && elems[j] > key) {
                elems[j + 1] = elems[j];
                j--;
            }
            elems[j + 1] = key;
        }
    }
```

a) How many test cases are required to get proper coverage of the constructor? Explain your answer and also give a specific set of test inputs that would give proper coverage.

> The input $[1]$ executes the outer loop zero times. The input $[1, 2]$ executes it one time and executes the inner loop zero times. The input $[2, 1]$ also executes the outer loop one time but now the inner loop executes once. Finally, the input $[3, 2, 1]$ executes the outer loop twice and the inner loop twice. This is 4 cases in total.