Quiz Section 5: Mutation

Task 1 – Two Sides of the Same Join [12 pts]

We plan to provide the following method:

```
/**
 * Join the two given lists into a single one
 * @requires first != null, second != null
 * ...
 */
public static List<Integer> join(List<Integer> first, List<Integer> second);
```

To do so, we need to fill in the rest of the specification.

We are considering the following alternatives:

```
@return first ++ second                                    // Spec A

@modifies first                                            // Spec B
@return first ++ second

@modifies first, second                                    // Spec C
@return first ++ second

@modifies first                                            // Spec D
@effects first = first_0 ++ second
@return first_0 ++ second

@modifies first, second                                    // Spec E
@effects first = first_0 ++ second
@return a list
```

a) Fill in the following table explaining the relationships between each pair of specifications. Write an "S" for if the spec on left (the row) is stronger than the name on top (the column), a "W" if it is weaker, and a "—" if they are incomparable.

	A	**B**	**C**	**D**	**E**
A	X				
B		X			
C			X		
D				X	
E					X

b) Not every combination of `@modifies`, `@effects`, and `@return` behaviors appearing in the specifications on the previous page would be sensible. For example, consider the following specification:

```
@effects first = first_0 ++ second
@return second
```

What is wrong with this specification? Why shouldn't we use it?

Task 2 – Test, Ice, Compression, Elevation [13 pts]

In this problem, we will write tests for various `join` functions which will also appear on the homework.

a) First, consider a version of `join`, which does not mutate either argument:

```
/**
 * Join the two given lists into a single one
 * @requires first != null, second != null
 * @returns first ++ second
 */
public static List<Integer> join(List<Integer> first, List<Integer> second) {
    List<Integer> newList = new ArrayList<>();
    newList.addAll(first);
    newList.addAll(second);
    return newList;
}
```

Fill in the missing parts of the following JUnit test for this version of `join`.

```
    @Test
    public void testJoin() {
        List<Integer> list1 = Arrays.asList(new int[] { 1, 2 });
        List<Integer> list2 = Arrays.asList(new int[] { 3, 4 });
        assertEquals(_____________________________________, join(list1, list2));

        List<Integer> list3 = Arrays.asList(new int[] { 1 });
        List<Integer> list4 = Arrays.asList(new int[] { 2, 3, 4 });
        assertEquals(_____________________________________, join(list3, list4));
    }
```

b) Next, consider the following version of `join`, which mutates `first` and does not return anything.

```
/**
 * Join the two given lists into a single one
 * @requires first != null, second != null
 * @modifies first
 * @effects first = first_0 ++ second
 */
public static void join(List<Integer> first, List<Integer> second) {
    first.addAll(second);
}
```

Rewrite the JUnit test above to use this new definition of `join` on the same inputs as above.

c) This version should be longer than before. Why is that the case?

d) Finally, consider the version of `join`, which modifies both `first` and `second`.

```
/*
 * Join the two given lists into a single one
 * @requires first != null, second != null
 * @modifies first, second
 * @effects first = first_0 ++ second_0, second = nil
 * @return first
 */
public static List<Integer> join(List<Integer> first, List<Integer> second) {
    while (!second.isEmpty()) {
      first.add(second.get(0));
      second.remove(0);
    }
    return first;
}
```

Rewrite the JUnit test again to properly test this new definition of `join`.

e) Imagine we replaced the spec for `join` in part d with the following spec:

```
/*
 * Join the two given lists into a single one
 * @requires first != null, second != null
 * @modifies first, second
 * @effects first = first_0 ++ second
 * @return first
 */
```

What would we need to change about our test cases?

The next problem concerns the following ADT:

```
/**
 * Represents a **mutable** collection of integers.
 *
 * Clients can think of a set as a list of integers that contains no duplicates.
 * The order of the integers is important and the "pop" operation promises
 * to remove the first element in the list.
 */
public class MutableIntSet {
  /**
   * Determines whether n is in the list.
   * @param n the number to look for in the list
   * @returns contains(obj, n), where
   *     contains(nil, n)    := false
   *     contains(m :: L, n) := true           if m = n
   *     contains(m :: L, n) := contains(n, L) if m != n
   */
  public boolean contains(int n);

  /**
   * Adds n to the list if not already present.
   * @param n the number to add to the new list.
   * @modifies obj
   * @effects obj = add(n, obj_0), where
   *     add(n, L) := L       if contains(n, L)
   *     add(n, L) := n :: L  if not contains(n, L)
   */
  public void add(int n);

  /** Removes and returns the first element in the collection. .... */
  public int pop();
}
```

Task 3 – Good News and Add News [10 pts]

Answer the following questions about the specification of `MutableIntSet`. Assume that `T` is an instance of this class whose abstract state is $1 :: 2 :: 3 ::$ nil.

a) Would `T.add(3)` actually change `obj`? If not, why is that allowed when it says `@modifies obj`.

b) Now, consider a call `T.add(4)`. Explain how the operation of `MutableIntSet.add` differs from that of `IntSet.add` from Homework 3.

c) What is the abstract state of T after the following code[1]:

```
T.add(4);
T.add(2);
T.add(0);
```

d) Write a specification for the method `pop`. It should return the head of the list and change the abstract state to be the tail of the list.

[1] This is forward reasoning.