Homework 3

Due: Wednesday, April 29th, 11:59pm

Task 1 – Splicing in a Spec [12 pts]

Consider the `splice` method:

```
/**
 * Return the list obtained by inserting all elements of second into first
 * at index first.size() / 2
 * @requires first != null, second != null,
 *             first and second must not refer to the same object
 * ...
 */
public static List<Integer> splice(List<Integer> first, List<Integer> second);
```

Before we implement the method, we need to complete the specification. We will consider the following alternatives:

```
@return splice(first, second)                                    // Spec A

@modifies first                                                  // Spec B
@return splice(first_0, second)

@modifies first, second                                          // Spec C
@return splice(first_0, second_0)

@modifies first                                                  // Spec D
@effects first = splice(first_0, second)
@return splice(first_0, second)

@modifies first, second                                          // Spec E
@effects first = splice(first_0, second_0)
@return a list
```

where `splice(first, second)` is the list `xs ++ second ++ ys`, where `first = xs ++ ys` and `|xs| = first.size() / 2`.

a) In the following questions, decide how the two specs compare, and explain your reasoning briefly in 1-2 sentences.

- **i.** Specs **A** and **B**.
- **ii.** Specs **B** and **D**.
- **iii.** Specs **A** and **D**.

b) For each of the following implementations, state which of the specifications A-E it satisfies, or none. If it does not satisfy some specification, briefly explain why.

(a)
```
public static List<Integer> splice(List<Integer> first, List<Integer> second) {
    List<Integer> result = new ArrayList<>();
    int mid = first.size() / 2;
    for (int i = 0; i < mid; i++) {
        result.add(first.get(i));
    }
    result.addAll(second);
    for (int i = mid; i < first.size(); i++) {
        result.add(first.get(i));
    }
    return result;
}
```

(b)
```
public static List<Integer> splice(List<Integer> first, List<Integer> second) {
    int i = first.size() / 2;
    while (!second.isEmpty()) {
        first.add(i, second.remove(0));
        i++;
    }
    return first;
}
```

(c)
```
public static List<Integer> splice(List<Integer> first, List<Integer> second) {
    List<Integer> result = new ArrayList<>();
    int mid = first.size() / 2;
    for (int i = 0; i < first.size(); i++) {
        result.add(first.get(i));
        if (i == mid) {
            result.addAll(second);
        }
    }
    return result;
}
```

Task 2 – In-dex we trust [6 pts]

In this task, you will practice **forward** and **backward** reasoning on method calls.

Fill in each blank without weakening the assertion. Use "↔" between an assertion and an equivalent simplification.

For part (a), use the following specification of the static method `indexOf`:

```
/**
 * @param xs the list to search (read-only for this specification)
 * @param key the value to find
 * @requires key occurs at least once as a value in xs
 * @return the smallest index i such that xs.get(i) == key
 */
public static int indexOf(List<Integer> xs, int key);
```

The first assertion in the code segment for part (a) gives the abstract state of A.

a) Use **forward** reasoning to fill in P. Then explain briefly (in one sentence) why P implies Q.

$\{\!\{ A = 1 :: 4 :: 9 :: \text{nil and } k = 4 \}\!\}$
`i = indexOf(A, k);`
$\{\!\{ P : \underline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad} \}\!\}$
$\{\!\{ Q : \ i = 1 \text{ and } k = 4 \}\!\}$

b) We define `sum` as the function that computes the sum of all elements in a list, and `list.add(0, x)` prepends `x` to `list`, updating the abstract state from $\texttt{list}_0$ to $\texttt{x::list}_0$. You may use $\text{sum}(x :: L_0) = x + \text{sum}(L_0)$ without proof.

The first assertion in the code segment for part (b) gives the abstract state of `nums`.

Use **backward** reasoning to fill in Q. Then explain briefly (in one sentence) why P implies Q.

$\{\!\{ P : \ L_0 = 2 :: 3 :: \text{nil and } t = 7 \}\!\}$
$\{\!\{ Q : \underline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad} \}\!\}$
`nums.add(0, t);`
$\{\!\{ \text{sum}(L) > 10 \}\!\}$

The next few problems concern the following ADT:

```
/**
 * Represents an *immutable* map with integer keys and values.
 * Think of this as a list of integer pairs (an association list).
 */
public interface IntMap {
  /**
   * Creates and returns a new map with k mapping to v. The new map
   * should contain all existing mappings in obj when the key is not
   * equal to k.
   * @param k the key to add or update
   * @param v the value for key k
   * @return the new map described above
   */
  public IntMap put(int k, int v);

  /**
   * Returns the corresponding value of key k
   * @param k the key whose value is to be retrieved
   * @requires obj to contain k
   * @return v the value for key k
   */
  public int get(int k);

  /**
   * Returns whether the map contains a mapping for key k.
   * @param k the key to check
   * @return true if the mapping exists, false otherwise
   */
  public boolean contains(int k);
}
```

Task 3 – Unlock Your True Put-tential [12 pts]

We will represent the `IntMap` ADT from above as `IntMapImpl`. The keys and values will be stored in two lists. These fields and an RI are shown below:

```
public class IntMapImpl implements IntMap {
    // RI: length of keys is equal to the length of values
        ...
    private int[] keys;
    private int[] values;

    private IntMapImpl(int[] keys, int[] values) {
        this.keys = keys;
        this.values = values;
    }
}
```

Note that the constructor is private! Methods in the class will still be able to call this constructor, but the caller is expected to maintain the AF/RI with the input `keys` and `values`.

Le Benoit has taken 331 and wants to add an AF and RI (**in addition to the existing RI**). Consider the following AFs and RIs for `IntMapImpl`.

- 1:

```
// RI: keys has no duplicates
// AF: obj is the map of keys to their corresponding (same index)
       values
```

- 2:

```
// RI: keys has no duplicates and is sorted
// AF: obj is the map of keys to their corresponding (same index)
       values
```

- 3:

```
// RI: None
// AF: obj is the map of keys to their corresponding (same index)
       values. If there are duplicate keys, take the rightmost key
       and value
```

For each of the following implementations of `put`, state the AF/RI(s) (1–3) for which the implementation would satisfy the specification of the `put` method in our provided ADT. In each case, briefly explain why.

Hint: `Arrays.binarySearch(int[] arr, int key)` requires that the array `arr` be sorted prior to the call. If there are duplicates, then there is no guarantee which one is found. Look at the online javadocs for more information!

Hint: `System.arraycopy(int[] src, int srcPos, int[] dest, int destPos, int length)` copies an array from `src`, beginning at `srcPos`, to the array `dest` at `destPos`. `length` elements are copied.

a)
```
public IntMap put(int k, int v) {
    int len = this.keys.length;
    int[] newKeys = new int[len + 1];
    int[] newValues = new int[len + 1];
    System.arraycopy(this.keys, 0, newKeys, 0, len);
    System.arraycopy(this.values, 0, newValues, 0, len);
    newKeys[len] = k;
    newValues[len] = v;
    return new IntMapImpl(newKeys, newValues);
}
```

b)
```
public IntMap put(int k, int v) {
    int len = this.keys.length;
    int index = -1;
    for (int i = len - 1; i >= 0; i--) {
        if (this.keys[i] == k) {
            index = i;
        }
    }

    if (index == -1) {
        int[] newKeys = new int[len + 1];
        int[] newValues = new int[len + 1];
        System.arraycopy(this.keys, 0, newKeys, 0, len);
        System.arraycopy(this.values, 0, newValues, 0, len);
        newKeys[len] = k;
        newValues[len] = v;
        return new IntMapImpl(newKeys, newValues);
    } else {
        int[] newValues = new int[len];
        System.arraycopy(this.values, 0, newValues, 0, len);
        newValues[index] = v;
        return new IntMapImpl(this.keys, newValues);
    }
}
```

c)
```
public IntMap put(int k, int v) {
    int len = this.keys.length;
    int index = -1;
    for (int i = 0; i < len; i++) {
        if (this.keys[i] == k) {
            index = i;
        }
    }

    if (index == -1) {
        int[] newKeys = new int[len + 1];
        int[] newValues = new int[len + 1];
        System.arraycopy(this.keys, 0, newKeys, 0, len);
        System.arraycopy(this.values, 0, newValues, 0, len);
        newKeys[len] = k;
        newValues[len] = v;
        return new IntMapImpl(newKeys, newValues);
    } else {
        int[] newValues = new int[len];
        System.arraycopy(this.values, 0, newValues, 0, len);
        newValues[index] = v;
        return new IntMapImpl(this.keys, newValues);
    }
}
```

d)
```
public IntMap put(int k, int v) {
    int len = this.keys.length;
    int index = Arrays.binarySearch(this.keys, k);
    if (index >= 0) {
        int[] newValues = new int[len];
        System.arraycopy(this.values, 0, newValues, 0, len);
        newValues[index] = v;
        return new IntMapImpl(this.keys, newValues);
    } else {
        int insert = -(index + 1);

        int[] newKeys = new int[len + 1];
        int[] newValues = new int[len + 1];

        System.arraycopy(this.keys, 0, newKeys, 0, insert);
        System.arraycopy(this.values, 0, newValues, 0, insert);

        newKeys[insert] = k;
        newValues[insert] = v;
```

```
        System.arraycopy(this.keys, insert, newKeys, insert + 1, len - insert);
        System.arraycopy(this.values, insert, newValues, insert + 1, len - insert);

        return new IntMapImpl(newKeys, newValues);
    }
}
```

Task 4 – What does this candidate bring to the mu-table? [6 pts]

In this task, you will define a MutableIntMap interface by rewriting the immutable IntMap interface from the previous problems.

For each of the three functions from IntMap, you will provide both an updated specification and function signature for the mutable version that follows 331 standards. If the immutable and mutable versions of the function have both the same spec and same signature, write NO CHANGE.

a) `put`

b) `get`

c) `contains`

Task 5 – X-Map [12 pts]

Faraz still has aspirations for his trillion dollar startup. To reach his goal, he continues Le Benoit's `IntMapImpl`. However, after the cybersecurity mishap from last week's homework, he is hesitant to share implementation details with you. Instead, he directs you to the following AF and RI pairs, courtesy of Le Benoit:

```
// RI: length of keys is equal to the length of values.
// AF: obj is the map of keys to their corresponding (same index) values.
//     If there are duplicate keys, take the rightmost key and value.
```

```
// RI: length of keys is equal to the length of values and keys has no
//     duplicates and is sorted.
// AF: obj is the map of keys to their corresponding (same index) values.
```

a) Faraz has the following concrete state for an `IntMap` called `map`:

```
this.keys = {0, 1, 3, 7};
this.values = {1, 4, 2, 8};
```

He then performs the following sequence of method calls:

```
map = map.put(6, 4);
map = map.put(7, 4);
map = map.put(9, 4);
map = map.put(Integer.MAX_VALUE, Integer.MIN_VALUE);
```

Describe possible concrete states of `map` after the code above gets executed for both of the above RI and AF pairs. Note that the final abstract state is the same for both RI and AF pairs. Describe the final abstract state and explain briefly what allows the abstract states to be the same but the concrete states to be different.

b) Faraz settles on the following AF and RI (thanks again, Le Benoit!):

```
// RI: length of keys is equal to the length of values and keys has no
//     duplicates and is sorted.
// AF: obj is the map of keys to their corresponding (same index) values.
```

Before incorporating your immutable map ADT into the codebase for his trillion dollar startup, Faraz proposes adding a `remove` method to `IntMap`:

```
/**
 * Returns a new map containing all key-value pairs except for
 * the mapping for k if such a mapping exists in obj. Otherwise
 * return obj
 * @param k The key to search for and remove.
 * @return the new map described above
 */
public IntMap remove(int k);
```

For secret reasons, Faraz has another map which he cleverly calls `map` that has the following concrete state:

```
this.keys = {0, 1, 3, 6, 7, 9, Integer.MAX_VALUE};
this.values = {1, 4, 2, 4, 4, 4, Integer.MIN_VALUE};
```

He then executes the following sequence of method calls:

```
map = map.remove(Integer.MIN_VALUE);
map = map.remove(0);
map = map.remove(6);
map = map.remove(8);
```

Describe possible concrete states of `map` after the code above gets executed for the given RI and AF pair.

c) After learning about the many benefits (and dangers) of mutability, Faraz decides to implement a mutable version too. Since he's feeling pretty lazy, he copies the original immutable spec and minorly tweaks it:

```
/**
 * Removes the key-value pair corresponding to k (if it exists).
 * @param k The key to search for and remove.
 * @modifies obj
 * @effects obj = obj_0 without the key-value pair corresponding to k
 *                (or just obj_0 if no such key-value pair exists).
 * @return obj (as a reference to this)
 public MutableIntMap remove(int k);
```

Notice how he is mixing mutators and producers! Afterwards, he constructs immutable and mutable maps called `imap1` and `mmap1`, respectively, from the following concrete state:

```
this.keys = {1, 4, 7, 9, 10, 13};
this.values = {999, 6, 0, -1, 10, 14};
```

Then he runs the following snippet of code:

```
imap2 = imap1.remove(9);
imap3 = imap2.remove(13);

mmap2 = mmap1.remove(9);
mmap3 = mmap2.remove(13);
```

Describe the abstract states for all six maps (three for both immutable and mutable) after the above code has been executed.