Homework 6

Due: Friday, November 7th, 6pm

Task 1 – Cool Your Sets [6 pts]

In Homework 3, we looked at an immutable `IntSet`. The class below implements a mutable set of integers. It also has a few extra operations.

```
public class MutableIntSet {
    // AF: obj = this.elems
    // RI: this.elems is sorted (increasingly) and has no dups
    private List<Integer> elems;

    // Creates a set with the given list.
    public MutableIntSet(List<Integer> elems) {
      this.elems = sortedNoDups(elems);
    }

    // Checks if the set contains the given.
    public boolean contains(int n) {
        return Collections.binarySearch(this.elems, n) >= 0;
    }

    // Returns a list containing all the items in the set.
    public List<Integer> items() {
        return this.elems;
    }

    // Continued on the next page...
```

```
    // Adds all the given elements into the set.
    public void addAll(MutableIntSet other) {
        int n = this.elems.size();
        int m = other.elems.size();

        // If all the new elements are smaller or larger than all of the
        // current elements, we can skip a re-sort of the whole thing
        // since this.elems and other.elems are respectively sorted
        if (n > 0 && m > 0 && this.elems.get(n-1) < other.elems.get(0)) {
            this.elems.addAll(other.elems);
        } else if (n > 0 && m > 0 &&
                   other.elems.get(m-1) < this.elems.get(0)) {
            List<Integer> newElems = other.elems;
            newElems.addAll(this.elems);
            this.elems = newElems;
        } else {
            this.elems.addAll(other.elems);
            sortedNoDups(this.elems);  // expensive re-sort
        }
    }

    // Returns the list without any duplicates and in sorted order
    private static List<Integer> sortedNoDups(List<Integer> vals) {
        Collections.sort(vals);

        for (int i = vals.size() - 1; i > 0; i++) {
          if (vals.get(i-1) == vals.get(i))
            vals.remove(i);
        }

        return vals;
    }
}
```

a) In light of what we learned this week, what problems do you see with this **implementation**? We're looking for problems with how the ADT was implemented, not issues with the specifications. Ignore efficiency; just focus on correctness.

b) Explain how to fix the issues described in (a).

c) Did `IntSet` from Homework 3, which was **immutable**, have any of these same problems?

In the remaining problems, we will use the ADT from this week's section:

```
/**
 * A **mutable** list of integers that can only be modified by adding
 * and removing elements at the front, or emptying the entire list.
 */
public interface IntStack {
    /**
     * Returns the length of the list.
     * @returns len(obj)
     */
    int length();

    /**
     * Adds the given number to the front of the list.
     * @param n The number to add to the list.
     * @modifies obj
     * @effects obj = n :: obj_0
     */
    void push(int n);

    /**
     * Removes and returns the first element in the list.
     * @requires len(obj) /= 0
     * @modifies obj
     * @effects obj_0 = n :: obj
     * @returns n
     */
    int pop();

    /**
     * Removes all elements in the list.
     * @modifies obj
     * @effects obj = nil
     */
    void clear();
}
```

Just as we did in section, we will implement this interface by storing the list in a compressed manner, where multiple consecutive and identical values are stored as a pair that records the value and how many times it occurs.

To formalize this, we first need a type that represents a *list* of *pairs* of integers. We can define such a type inductively as follows:

$$\textbf{type } \mathsf{PList} := \mathsf{pnil} \mid \mathsf{pcons}((\mathbb{Z}, \mathbb{N}), \mathsf{PList})$$

We will use the shorthand "::", when applied to a PList, to refer to a pcons operation. For example, the expression "$(1,2) :: (3,4) ::$ pnil" is shorthand for $\mathsf{pcons}((1,2),\ \mathsf{pcons}((3,4),\ \mathsf{pnil}))$. Hopefully, this will not cause confusion with cons on List.

With that definition in hand, our concrete representation will store the list $1 :: 1 :: 1 :: 2 :: 3 :: 3 ::$ nil, which contains runs of 1s and 3s, as the shorter list $(1,3) :: (2,1) :: (3,2) ::$ pnil. The following class uses this concrete representation:

```
public class CompressedIntStack implements IntStack {
    // AF: obj = expand(this.pairs)
    private PairList pairs;
```

This uses the `PairList` class, defined below, to store a PList, and the function expand : (PList) $\to$ List, which is defined as follows:

$$\begin{aligned} \mathsf{expand}(\mathsf{pnil}) &:= \mathsf{nil} \\ \mathsf{expand}((n,0) :: L) &:= \mathsf{expand}(L) \\ \mathsf{expand}((n,c+1) :: L) &:= n :: \mathsf{expand}((n,c) :: L) \end{aligned}$$

The abstraction function of `CompressedIntStack` says that the abstract state is the list that you would get by expanding the PList stored in the field `pairs`.

Finally, the `PairList`, which stores a PList directly as a linked list, is defined as follows:

```
/** Represents a list of pairs. */
private static class PairList {
    public final int value;
    public final int count;
    public final PairList next;

    public PairList(int value, int count, PairList next) {
        this.value = value;
        this.count = count;
        this.next = next;
    }
}
```

Task 2 – It's Raining Len [8 pts]

Prove that $\mathsf{len}(\mathsf{expand}((n, m) :: L)) = m + \mathsf{len}(\mathsf{expand}(L))$ holds by induction on m.

As a reminder, the function $\mathsf{len} : (\mathsf{List}) \to \mathbb{N}$ is defined by:

$$\mathsf{len}(\mathsf{nil}) := 0$$
$$\mathsf{len}(x :: L) := 1 + \mathsf{len}(L)$$

Task 3 – I Would Never Loop So Low [12 pts]

The `length` method in `CompressedIntStack` is implemented as follows:

```
public int length() {
  int n = 0;
  PairList curr = this.pairs;
  {{ P1: _____ }}
  {{ Inv: n + len(expand(curr)) = len(expand(this.pairs)) }}
  while (curr != null) {
    {{ P2: _____ }}
    {{ Q2: _____ }}
    n = n + curr.count;
    curr = curr.next;
  }
  {{ P3: _____ }}
  {{ Q: len(obj) = n }}
  return n;
}
```

a) Use forward reasoning to fill in P_1. Then prove that this implies that the invariant holds initially.

b) Use forward reasoning to fill in P_2 and backward reasoning to fill in Q_2. Then, prove P_2 implies Q_2.

Note that curr $\neq$ pnil means that curr $=$ (curr.value, curr.count) :: curr.next.

Feel free to use the fact you proved in Task 2 here.

c) Use forward reasoning to fill in P_3. Then, prove that this implies the spec's claim that $n = \text{len}(\text{obj})$.

Task 4 – Extra Credit: Bop to the Pop [10 pts]

The pop method in `CompressedIntStack` is implemented as follows:

```
public int pop() {
  if (this.pairs == null) {
    throw new IllegalArgumentException("can't pop from empty stack");
  }
 {{ this.pairs_0 = (n, c) :: L }}
  int removed = this.pairs.value;
  if (this.pairs.count == 1) {
    {{ P_1: _____ }}
    this.pairs = this.pairs.next;
    {{ P_2: _____ }}
  } else {
    {{ P_3: _____ }}
    this.pairs.count = this.pairs.count -1;
    {{ P_4: _____ }}
  }
  return removed;
}
```

a) Use forward reasoning to fill in P_1 and P_2.

b) Prove that P_2 implies the spec's `@effects` tag.

c) Use forward reasoning to fill in P_3 and P_4.

d) Prove that P_4 implies the spec's `@effects` tag.