CSE 331: Software Design & Implementation Winter 2026

Homework 6

Due: Wednesday, February 25th, 11:59pm

In the section worksheet, we introduced PLists, which are lists of pairs of numbers, and used them to compress lists. We defined PLists inductively as such:

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

In this homework, we show another use case of PLists: to represent a **map**. We interpret $(1, 2) :: (3, 4) :: \text{pnil}$ as mapping the key 1 to the value 2 and the key 3 to the value 4. This is often called an "association list", with each pair being an association of a key to a value.

We can formalize this by defining a function get-value : $(\text{PList}, \mathbb{Z}) \to \mathbb{Z}$ that will return the value associated with a given key. It can be defined recursively as follows:

$$\begin{aligned} \text{get-value}((x, v) :: L,\ y) &:= v && \text{if } x = y \\ \text{get-value}((x, v) :: L,\ y) &:= \text{get-value}(L,\ y) && \text{if } x \neq y \end{aligned}$$

For example, we can see that

$$\begin{aligned} &\text{get-value}((1, 2) :: (3, 4) :: \text{pnil}, 3) \\ &= \text{get-value}((3, 4) :: \text{pnil}, 3) && \text{def of get-value (since } 1 \neq 3) \\ &= 4 && \text{def of get-value (since } 3 = 3) \end{aligned}$$

Note that the value of get-value(L, y) is undefined if y is not a key in the list. Hence, before using get-value, we must know whether a given value is a key in the list...

We can find out if y is a key in the list using the function contains-key : $(\text{PList}, \mathbb{Z}) \to \mathbb{B}$, defined by

$$\begin{aligned} \text{contains-key}(\text{nil},\ y) &:= \text{false} \\ \text{contains-key}((x, v) :: L,\ y) &:= \text{true} && \text{if } x = y \\ \text{contains-key}((x, v) :: L,\ y) &:= \text{contains-key}(L,\ y) && \text{if } x \neq y \end{aligned}$$

For example, we can see that

$$\begin{aligned} &\text{contains-key}((1, 2) :: (3, 4) :: \text{pnil}, 3) \\ &= \text{contains-key}((3, 4) :: \text{pnil}, 3) && \text{def of contains-key (since } 1 \neq 3) \\ &= \text{true} && \text{def of contains-key (since } 3 = 3) \end{aligned}$$

Additionally, we define the function update-key : $(\text{PList}, \mathbb{Z}, \mathbb{Z}) \to \text{PList}$, as

$$\begin{aligned} \text{update-key}(\text{nil},\ x,\ y) &:= \text{nil} \\ \text{update-key}((k, v) :: L,\ x,\ y) &:= (k, y) :: \text{update-key}(L,\ x,\ y) && \text{if } k = x \\ \text{update-key}((k, v) :: L,\ x,\ y) &:= (k, v) :: \text{update-key}(L,\ x,\ y) && \text{if } k \neq x \end{aligned}$$

Now consider the interface `IntMap` on the next page.

```
/**
 * Represents a *mutable* map with integer keys and values.
 * Think of this as a list of integer pairs (an association list).
 */
public interface IntMap {
  /**
   * Adds or replaces the mapping for k so it maps to v.
   * @param k the key to add or update
   * @param v the value for key k
   * @modifies obj
   * @effects obj = (k, v) :: obj_0 if !contains-key(obj, k)
               else obj = update-key(obj_0, k, v)
   */
  public void put(int k, int v);

  /**
   * Returns the corresponding value to key k
   * @param k the key whose value is to be retrieved
   * @requires contains-key(obj, k)
   * @return get-value(obj, k)
   */
  public int get(int k);

  /**
   * Returns whether the map contains a mapping for key k.
   * @param k the key to check
   * @return contains-key(obj, k)
   */
  public boolean contains(int k);

  /**
   * Removes all entries from the map.
   * @modifies obj
   * @effects obj = nil.
   */
  public void clear();
}
```

We will consider implementing `IntMap` using the following class, which stores the keys and values in two separate lists, which must be of the same length:

```
public class ListPairMap implements IntMap {
    // RI: len(this.keys) = len(this.vals) and this.keys has no duplicates
    // AF: obj = zip(this.keys, this.vals);
    private List keys;
    private List vals;
}
```

The abstraction function uses the function zip : (List, List) $\rightarrow$ PList, which turns two lists of the same length into a list of pairs. It is defined recursively by:

$$\begin{aligned} \mathsf{zip}(\mathsf{nil}, \mathsf{nil}) &:= \mathsf{pnil} \\ \mathsf{zip}(k :: L, v :: R) &:= (k, v) :: \mathsf{zip}(L, R) \end{aligned}$$

For example, we can see that

$$\begin{aligned} &\mathsf{zip}(1 :: 2 :: \mathsf{nil}, 3 :: 4 :: \mathsf{nil}) \\ &= (1,3) :: \mathsf{zip}(2 :: \mathsf{nil}, 4 :: \mathsf{nil}) && \text{def of zip} \\ &= (1,3) :: (2,4), \mathsf{zip}(\mathsf{nil}, \mathsf{nil}) && \text{def of zip} \\ &= (1,3) :: (2,4) :: \mathsf{pnil} \end{aligned}$$

The function zip is undefined when the two lists have different lengths. This is fine in our case because the RI requires the list of keys and list of values to have equal length. For example, if we have this.keys $= [1, 2]$ and this.vals $= [3, 4]$, then the RI is satisfied and the abstract state is the association list $(1, 2) :: (3, 4) ::$ pnil.

The `List` type used above is just our standard representation of integer lists:

```
private static class List {
    public final int hd;
    public final List tl;

    public List (int hd, List tl) {
        this.hd = hd;
        this.tl = tl;
    }
}
```

Task 1 – Map, Crackle, and Pop [12 pts]

For each of the following constructor/method implementations, state whether the following are satisfied, and if not, *briefly* explain **why** that part was broken:

1. the specification of the method or constructor
2. the RI of `ListPairMap`
3. safe implementation practices according to 331 style

a) Consider the following implementation of the constructor. (`java.util.List<Integer>` refers to the standard library list interface in Java. Internally, the implementation uses `List`, which is our simple singly linked list class.)

```
/**
    * Creates a new ListPairMap given a list of the keys and
    * a list of the values those keys map to
    * @param keys, the keys of the map
    * @param vals, the values of the map
    * @requires len(keys) = len(values)
    * @effects obj = zip(keys, values)
    */
public ListPairMap(java.util.List<Integer> keys,
                   java.util.List<Integer> vals) {
    if (keys.size() != vals.size()) {
        throw new IllegalArgumentException("arguments must have same length");
    }
    this.keys = null;
    this.vals = null;

    for (int i = keys.size() - 1; i >= 0; i--) {
        this.keys = new List(keys.get(i), this.keys);
        this.vals = new List(vals.get(i), this.vals)
    }
    checkRep();
}
```

You may assume `checkRep()` is defined correctly. (It throws an exception if the RI is not true.)

b) Consider the following implementation of `get`:

```
public int get(int k) {
    if (!contains(k)) {
        throw new IllegalArgumentException("map must contain k");
    }
    List ks = this.keys;
    List vs = this.vals;
    while (ks != null && ks.hd != k) {
        ks = ks.tl;
        vs = vs.tl;
    }
    return vs.hd;
}
```

c) Consider the following implementation of `put`:

```
public void put(int k, int v) {
    this.keys = new List(k, this.keys);
    this.vals = new List(v, this.vals);
    checkRep();
}
```

d) Consider the following implementation of `clear`:

```
public void clear() {
    List clearedKeys = null;
    List clearedVals = null;
}
```

Task 2 – Guilty of High Reason [15 pts]

Consider this implementation of `contains`. Remember to highlight your assertions.

```
public boolean contains(int k) {
  List ks = this.keys;
  List vs = this.vals;
  {{ P1: _____ }}
  {{ Inv: contains-key(obj, k) = contains-key(zip(ks, vs), k) }}
  while (ks != null && vs != null) {
    {{ Inv and ks = ks.hd :: ks.tl and vs = vs.hd :: vs.tl }}
    if (k == ks.hd) {
        {{ P2: _____ }}
        {{ Q2: contains-key(obj, k) = true }}
        return true;
    }
    {{ P3: Inv and ks = ks.hd :: ks.tl and vs = vs.hd :: vs.tl and k ≠ ks.hd }}
    {{ Q3: _____ }}
    ks = ks.tl;
    vs = vs.tl;
    {{ Inv: contains-key(obj, k) = contains-key(zip(ks, vs), k) }}
  }
  if (ks != null || vs != null) {
    throw new RuntimeException("RI does not hold");
  } else {
    return false;
  }
}
```

a) Use forward reasoning to fill in P_1.

b) Prove that P_1 implies that Inv holds initially.

c) Why does the RI not hold in the last `if` branch where it throws an exception? Explain *briefly*.

d) Use backward reasoning to fill in Q_3.

e) Use forward reasoning to fill in P_2.

f) Prove that P_2 implies Q_2.

For the remaining tasks, we will use the ADT introduced in 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 MutableIntStack {
    /**
     * 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();
}
```

In a similar manner to how we represented `Maps` as a list of pairs, we can have our concrete representation store the list $1 :: 1 :: 1 :: 2 :: 3 :: 3 ::$ nil as the shorter list $(1, 3) :: (2, 1) :: (3, 2) ::$ pnil. Essentially, we compress the runs of 1s and 3s to the second slot of the pair. As seen in section, we use the following concrete representation:

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

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`.

The function expand : (PList) $\rightarrow$ List is defined as follows:

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

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 3 – Goldilocks and the Three Pairs [12 pts]

The `length` method in `CompressedIntStack` is implemented as follows. Remember to highlight your final assertions.

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

a) Use forward reasoning to fill in P_2.

b) Use backward reasoning to fill in Q_2.

c) Prove that P_2 implies Q_2.
You may cite that $\text{len}(\text{expand}((n, m) :: L)) = m + \text{len}(\text{expand}(L))$ as Lemma 1.

d) Fill in the postcondition of the while loop in P_3.

e) Prove that P_3 implies the spec's claim, Q, that num = len(obj).

Task 4 – The Stack That Smiles Back, Goldfish! [12 pts]

The pop method in `CompressedIntStack` is implemented as follows, where we note a pair's value and count as (n, c), respectively, in math notation. Remember to highlight your final assertions. You may assume that $c > 0$, since if an integer exists in the stack, its count must be at least 1.

```
public int pop() {
  if (this.pairs == null) {
    throw new IllegalStateException("can't pop from empty stack");
  }
 {{ this.pairs_0 = (n, c) :: L }}
  int val = this.pairs.value;
  int runCount = this.pairs.count;
 {{ ____________________________________________ }}
  if (runCount == 1) {
    {{ ____________________________________________ }}
    this.pairs = this.pairs.next;
    {{ P_1: ____________________________________________ }}
  } else {
    {{ ____________________________________________ }}
    this.pairs = this.pairs.next;
    {{ ____________________________________________ }}
    this.pairs = new PairList(val, runCount -1, this.pairs);
    {{ P_2: ____________________________________________ }}
  }
  return val;
}
```

a) Use forward reasoning to fill in P_1.

b) Prove that P_1 implies the spec's `@effects` tag.

c) Use forward reasoning to fill in P_2.

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

Task 5 – Extra Credit: When Life Gives You Lemmas, Make Lemma-nade [0 pts]

In Task 3 we provided a Lemma... now prove it! Prove that the following holds by induction on m:

$$\mathsf{len}(\mathsf{expand}((n, m) :: L)) = m + \mathsf{len}(\mathsf{expand}(L))$$

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

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