
CSE 331

Software Design & Implementation

Spring 2023
Section 7 – Imperative Programming III

Administrivia

- HW7 released later today
 - Due Wednesday (5/17) @ 11:00pm
- Midterm on Friday (5/19) during usual class time

Mutable ADTs – Review

- Adding mutations can give us better efficiency
 - Saved memory
 - With arrays, gained ability to access any element
- However, with more mutations mean more complex reasoning
 - More facts to keep track of
 - More ways to make mistakes
 - More work to ensure we did it right
- With mutable heap state:
 - Must keep track of every alias that could mutate that state (**Rep Invariant**)
 - Previously, only needed to confirm RI held at end of constructor. Now we must ensure it holds for ANY method that could mutate the object

Avoiding Rep Exposure – Review

- Exposing your ADT to the client could potentially allow them to violate the Rep Invariant and break your code
- Using aliases increases risk of introducing unwanted bugs
- Options for avoiding rep exposure:
 1. **Copy In, Copy Out** – stores copies of mutable values passed to you and returns copies of not aliases to mutable state
 2. **Use immutable types** – lists are immutable, so you can freely accept and return them

Defensive Programming – Review

- Write code to check preconditions

```
// RI: 0 <= start < start+len <= vals.length
constructor(
    vals: number[], start: number, len: number) {
    if (start < 0) throw new Error(`bad start ${start}`);
    if (len < 0) throw new Error(`bad length ${len}`);
    if (vals.length < start + len)
        throw new Error(`${start+len} < ${vals.length}`);

    this.vals = vals;
    this.start = start;
    this.len = len;
};
```

- Can also write code to check postconditions
- Write code to check loop invariants and Rep Invariant (checkRep function)
 - Make the program crash if any of the invariants are violated (it is always better for your program to crash than run silently with a bug)

Question 1

Assume L and R are **sorted** and have **distinct** elements.

The call “with(L, R)” returns the elements that appear in either L or R (or both), combined in sorted order. That function is defined as follows:

```
func with([], R ++ [y])      := with([], R) ++ [y]
      with(L ++ [x], [])     := with(L, []) ++ [x]
      with(L ++ [x], R ++ [y]) := with(L ++ [x], R) ++ [y]  if  $x < y$ 
      with(L ++ [x], R ++ [y]) := with(L, R) ++ [x]         if  $x = y$ 
      with(L ++ [x], R ++ [y]) := with(L, R ++ [y]) ++ [x]  if  $x > y$ 
```

The call “without(L, R)” returns the elements of L that *do not* appear in R , in the same order as they came in L . That function is defined as follows:

```
func without([], R)        := []
      without(L ++ [x], []) := without(L, []) ++ [x]
      without(L ++ [x], R ++ [y]) := without(L ++ [x], R)      if  $x < y$ 
      without(L ++ [x], R ++ [y]) := without(L, R)              if  $x = y$ 
      without(L ++ [x], R ++ [y]) := without(L, R ++ [y]) ++ [x] if  $x > y$ 
```

- (a) Prove by induction that $\text{without}(L, []) = L$ for any array L .
- (b) Is it true that $\text{with}(L, []) = L$ for any array L ? Why or why not?
- (c) Is it true that $\text{with}([], R) = R$ for any array R ? Why or why not?

Question 1a

(a) Prove by induction that $\text{without}(L, []) = L$ for any array L .

(b) Is it true that $\text{with}(L, []) = L$ for any array L ? Why or why not?

(c) Is it true that $\text{with}([], R) = R$ for any array R ? Why or why not?

Question 2

(a) Prove by calculation that $\text{without}([1, 2, 3], [2, 4]) = [1, 3]$.

(b) The definition of “without” drops the last element $[y]$ from R on the recursive call when $x < y$ or $x = y$ but not when $x > y$. Suppose that we changed the rule when $x > y$ to the following:

$$\text{without}(L \uplus [x], R \uplus [y]) := \text{without}(L, R) \uplus [x]$$

so that $[y]$ is dropped in this case as well.

Show that we would then get the wrong answer for $\text{without}([1, 2, 3], [2, 4])$.

Question 2 continued

- (c) The recursive calls drop elements from the ends of the two arrays, so each recursive call before the last is of the form “without($L[0 .. i], R[0 .. j - 1]$)” or some integers $i, j \geq 0$. That is true both with the original definition and with the changed (incorrect) definition in part (b). There was something about the first definition that made it work, while the second definition did not.

What condition about how $L[i]$ relates to R on these recursive calls does the original definition guarantee that the changed definition does not?

Hint: have a look at where the calculation in (b) went off the rails.

Question 3

Prove by induction on S that, if $L[i]$ is less than every element of S (i.e., $L[i] < S[k]$ for $k = 0 \dots S.length - 1$) and both L and $R \# S$ are sorted and contain distinct elements, then we have

$$\text{without}(L[0 \dots i], R \# S) = \text{without}(L[0 \dots i], R)$$

Set ADT

- One use of a sorted array of distinct elements is to represent a **set**!
 - A **set** only cares about whether an element is present so when storing elements, so order does not matter
 - However, from the previous problems, sorted order provides benefits of calculating operations such as “with” and “without”
 - We can also define other set-like operations with this representation
- Notice that this is just an ADT!
 - The client would have the functionality of a **set** without ever knowing a list was used to implement it
- What are some ideas for the **Representation Invariant** for this ADT will be? What about **Abstract Function**?

Set Operations

Set – collection that only contains *unique* objects

Union ($A \cup B$)– a set containing all the elements of A or B (or both).

– Notice this is just the definition of “*with(L,R)*”!

– Ex:

$$A = [1, 2, 3] \quad B = [2, 4]$$

$$A \cup B = [1, 2, 3, 4]$$


Compliment (A') – a set containing all the elements of U that *does not* appear in A. The set U must also contain all the elements of A.

– Ex:

$$U = [1, 2, 3, 4] \quad A = [2, 4]$$

$$A' \text{ (in respect to } U) = [1, 3]$$

This is also the set difference of U in respect to A



Intersection ($A \cap B$) – the set of elements in A that are also in B

– Ex:

$$A = [1, 2, 3] \quad B = [2, 4]$$

$$A \cap B = [2]$$

Question 4

- (a) Given our set representation with a sorted list, what is an algorithm that we could use to determine whether a number exists in our set or not? (i.e. what can we use to implement the `contains()` function of our set)

- (b) The “intersection” of sets A and B contains the numbers that appear in both sets. How would we calculate the intersection if A and B are sorted arrays? (Hint: you’ll need more than one function call!)

- (c) The “complement” of a set A with respect to a set U (which must contain all the elements of A) is the set of numbers in U but not in A . How would we calculate the complement of A with respect to U if both A and U are sorted arrays?

- (d) Using your answers to (a-b), how could you calculate the union of A with the complement of B with respect to U (that contains both A and B)? Is there another way that we could write this as the complement of some (other) set?