CS9053, Tuesday Section
April 14, 2020
Due April 21, 2020, 11:55 PM
Prof. Dean Christakos

Part I: Working with stacks and queues

1. Write a method `remAllStack(Stack<Object> stack, Object item)` that takes a stack and an item, and that removes all occurrences of that item from the stack. After the removals, the remaining items should still be in the same order with respect to each other. For example, if you have a stack that contains (from top to bottom) {5, 2, 7, 2, 10}, if you use the method to remove all occurrences of 2, the resulting stack should contain (from top to bottom) {5, 7, 10}.

   Important guidelines:

- Your method may use either another stack or a queue to assist it. It may *not* use an array, linked list, or other data structure. When choosing between a stack or a queue, choose the one that leads to the more efficient implementation.
- You should assume that the method does *not* have access to the internals of the collection objects, and thus you can only interact with them using the methods in the interfaces that we discussed in lecture.

2. Write a method `remAllQueue(Queue<Object> queue, Object item)` that takes a queue and an item, and that removes all occurrences of that item from the queue. After the removals, the remaining items should still be in the same order with respect to each other. For example, if you have a queue that contains (from front to rear) {5, 2, 7, 2, 10} and you use the method to remove all occurrences of 2, the resulting queue should contain (from front to rear) {5, 7, 10}. The same guidelines that we specified for `remAllStack()` also apply here.

Part II: Sets

The reason we like Sets in Java is because they help us think about Sets in a mathematical sense and we can easily implement the functions of Sets that exist in Math—eg, Set intersections and unions. In Python, these set functions are explicit. In Java, they are not.

Create a class `MathSet` which extends `HashSet`. It should have three methods:

`public Set intersection(Set s2)`: Takes a Set, s2, and returns the intersection of the Set and s2—the elements that are in both sets.

`public Set union(Set s2)`: Takes a Set, s2, and returns the union of the Set and s2—the combination of all elements.

`public Set<Pair<T,S>> cartesianProduct(Set s2)`

I have provided a Pair class for this. Return the Cartesian Product of the base set, s and s2: $s \times s2$:

A **Cartesian product** of two sets A and B, written as $A \times B$, is the set containing **ordered** pairs from A and B. That is, if $C = A \times B$, then each element of C is of the form (x,y)
where $x \in A$ and $y \in B$:

$A \times B = \{(x,y) | x \in A \text{ and } y \in B\}$.

For example, if $A=\{1,2,3\}$ and $B=\{H,T\}$, then

$A \times B = \{(1,H),(1,T),(2,H),(2,T),(3,H),(3,T)\}$

Note that here the pairs are ordered, so for example, $(1,H) \neq (H,1)$. Thus $A \times B$ is **not** the same as $B \times A$.

Part III: Maps

A Map creates Key -> Value relationships. Each Key is unique, but different Keys may map to the same Values.

Start with a `Map<Object, Object>` and implement a static method “`getInverted`” that takes a Map object and returns a Map that reverses the original Map object and maps the Values to Keys. Since a single Value could have multiple Keys associated with it, the reversed Map will be of the type `Map<Object, Set<Object>>`