You want to be able to access the *largest element* in a stack.

Use the built-in `Stack` class to **implement a *new* class `MaxStack` with a function `getMax()` that returns the largest element in the stack.** `getMax()` should not remove the item.

Your stacks will contain only integers.

Gotchas

What if we push several items in increasing numeric order (like 1, 2, 3, 4...), so that there is a *new max* after each `push()`? What if we then `pop()` each of these items off, so that there is a *new max* after each `pop()`? Your algorithm shouldn't pay a steep cost in these edge cases.

You should be able to get a runtime of $O(1)$ for `push()`, `pop()`, and `getMax()`.

Breakdown

One lazy approach is to have `getMax()` simply walk through the stack and find the max element. This takes $O(n)$ time for each call to `getMax()`. But we can do better.

To get $O(1)$ time for `getMax()`, we could store the max integer as a member variable (call it `max`). But how would we keep it up to date?

For every `push()`, we can check to see if the item being pushed is larger than the current `max`, assigning it as our new `max` if so. But what happens when we `pop()` the current `max`? We could re-compute the current `max` by walking through our stack in $O(n)$ time. So our worst-case runtime for `pop()` would be $O(n)$. We can do better.

What if when we find a new current `max` (`newMax`), instead of overwriting the old one (`oldMax`) we held onto it, so that once `newMax` was popped off our stack we would know that our `max` was back to `oldMax`?

What data structure should we store our set of `max`s in? We want something where the last item we put in is the first item we get out ("last in, first out").

We can store our `max`s in another stack!

We can store our `maxs` in another stack.

Solution

We define *two* new stacks within our `MaxStack` class—`stack` holds all of our integers, and `maxesStack` holds our "maxima." We use `maxesStack` to keep our max up to date in constant time as we `push()` and `pop()`:

1. Whenever we `push()` a new item, we check to see if it's greater than or equal to the current max, which is at the top of `maxesStack`. If it is, we also `push()` it onto `maxesStack`.
2. Whenever we `pop()`, we also `pop()` from the top of `maxesStack` if the item equals the top item in `maxesStack`.

```
import java.util.Stack;
```

```java
public class MaxStack {

    private Stack<Integer> stack     = new Stack<>();
    private Stack<Integer> maxesStack = new Stack<>();

    // Add a new item to the top of our stack. If the item is greater
    // than or equal to the last item in maxesStack, it's
    // the new max! So we'll add it to maxesStack.
    public void push(int item) {
        stack.push(item);
        if (maxesStack.empty() || item >= maxesStack.peek()) {
            maxesStack.push(item);
        }
    }

    // Remove and return the top item from our stack. If it equals
    // the top item in maxesStack, they must have been pushed in together.
    // So we'll pop it out of maxesStack too.
    public int pop() {
        int item = stack.pop();
        if (item == maxesStack.peek()) {
            maxesStack.pop();
        }
        return item;
    }

    // The last item in maxesStack is the max item in our stack.
    public int getMax() {
        return maxesStack.peek();
    }
}
```

Complexity

$O(1)$ time for `push()`, `pop()`, and `getMax()`. $O(m)$ additional space, where m is the number of operations performed on the stack.

Notice that our time-efficient approach takes some additional space, while a lazy approach (simply walking through the stack to find the max integer whenever `getMax()` is called) took no additional space. We've traded some space efficiency for time efficiency.

What We Learned

Notice how in the solution we're *spending time* on `push()` and `pop()` so we can *save time* on `getMax()`. That's because we chose to optimize for the time cost of calls to `getMax()`.

But we could've chosen to optimize for something else. For example, if we expected we'd be running `push()` and `pop()` frequently and running `getMax()` rarely, we could have optimized for faster `push()` and `pop()` functions.

Sometimes the first step in algorithm design is *deciding what we're optimizing for*. Start by considering the expected characteristics of the input.