You're working with an intern that keeps coming to you with JavaScript code that won't run because the braces, brackets, and parentheses are off. To save you both some time, you decide to write a braces/brackets/parentheses validator.

Let's say:

- '(', '{', '[' are called "*openers*."
- ')', '}', ']' are called "*closers*."

Write an efficient function that tells us whether or not an input string's openers and closers are properly nested.

Examples:

- "{ [ ] ( ) }" should return `true`
- "{ [( ]) }" should return `false`
- "{ [}" should return `false`

Gotchas

Simply making sure each opener has a corresponding closer is not enough—we must also confirm that they are correctly *ordered*.

For example, "{ [( ]) }" should return `false`, even though each opener can be matched to a closer.

We can do this in $O(n)$ time and space. One iteration is all we need!

Breakdown

We can use a greedy approach to walk through our string character by character, making sure the string validates "so far" until we reach the end.

What do we do when we find an opener or closer?

Well, we'll need to keep track of our openers so that we can confirm they get closed properly. What data structure should we use to store them? **When choosing a data structure, we should start by deciding on the properties we want**. In this case, we should figure out how we will want to *retrieve* our openers from the data structure! So next we need to know: what will we do when we find a closer?

Suppose we're in the middle of walking through our string, and we find our first closer:

```
[ { ( ) ] . . . .
      ^
```

How do we know whether or not that closer in that position is valid?

A closer is valid if and only if it's the closer for the most recently seen, unclosed opener. In this case, '(' was seen most recently, so we know our closing ')' is valid.

So we want to store our openers in such a way that we can **get the most recently added one quickly**, and we can **remove the most recently added one quickly** (when it gets closed). Does this sound familiar?

What we need is a stack !

Solution

We iterate through our string, making sure that:

1. **each closer corresponds to the most recently seen, unclosed opener**
2. **every opener and closer is in a pair**

We use a stack to keep track of the most recently seen, unclosed opener. And if the stack is ever empty when we come to a closer, we know that closer doesn't have an opener.

So as we iterate:

- If we see an opener, we push it onto the stack.
- If we see a closer, we check to see if it is the closer for the opener at the top of the stack. If it is, we pop from the stack. If it isn't, or if the stack is empty, we return `false`.

If we finish iterating and our stack is empty, we know every opener was properly closed.

```ruby
require 'set'

def is_valid(code)
    openers_to_closers = {
        '(' => ')',
        '{' => '}',
        '[' => ']'
    }

    openers = Set.new(openers_to_closers.keys)
    closers = Set.new(openers_to_closers.values)

    openers_stack = []

    for i in 0...code.length
        char = code[i]
        if openers.include? char
            openers_stack.push(char)
        elsif closers.include? char
            if openers_stack.empty?
                return false
            else
                last_unclosed_opener = openers_stack.pop

                # if this closer doesn't correspond to the most recently
                # seen unclosed opener, short-circuit, returning false
                if openers_to_closers[last_unclosed_opener] != char
                    return false
                end
            end
        end
    end
    return openers_stack == []
end
```

Complexity

$O(n)$ time (one iteration through the string), and $O(n)$ space (in the worst case, all of our characters are openers, so we push them all onto the stack).

Bonus

In Ruby, sometimes expressions are surrounded by vertical bars, "|like this|". Extend your validator to validate vertical bars. Careful: there's no difference between the "opener" and "closer" in this case—they're the same character!

What We Learned

The trick was to use a stack.

It might have been difficult to have that insight, because you might not use stacks that much.

Two common uses for stacks are:

1. **parsing** (like in this problem)
2. **tree or graph traversal** (like depth-first traversal)

So remember, if you're doing either of those things, try using a stack!
