Grade 7/8 Math Circles

March 6/7/8/9, 2023

Recursion and Stack ADTs - Problem Set

Exercise Solutions

Exercise 1

Write a recursive function called `fibonacci` that takes a number `n` and gives the n^{th} term of the Fibonacci sequence. Keep in mind that the first two digits of the sequences are 1 and 1. If you need a reminder of how the Fibonacci sequence works, refer back to Example A.

Note: n^{th} means counting an arbitrary number, like $1^{st}, 2^{nd}, 3^{rd}, 4^{th}, ..., n^{th}$.

Solution

Since the first two digits of the sequence are 1 and 1, we actually have two base cases. If `n=1`, then we return 1, and if `n=2`, then we also return 1. So this means our function would look something like this.

```
1 fibonacci(n) {
2     if(n=1) {
3         return 1
4     } else if(n=2) {
5         return 1
6     } else {
7         recursive case
8     }
9 }
```

```
1 fibonacci(n) {
2     if(n=1 or n=2) {
3         return 1
4     } else {
5         recursive case
6     }
7 }
8
9
```

Note that these two ways of writing the function are the same, since in both base cases the return value is 1. The second way is a little more efficient to write since it uses less code. This would not work if the return values were different for the base cases.

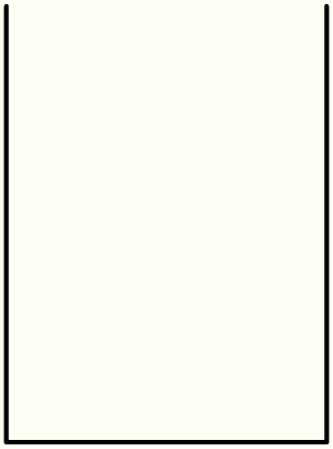
Next, we look at our recursive case. Since our n^{th} Fibonacci number is calculated by adding

the previous two Fibonacci numbers, we can deduce that our recursive case must be `return fibonacci(num-1) + fibonacci(num-2)`. So we get the following function

```
1 fibonacci(n) {
2     if (n=1 or n=2) {
3         return 1
4     } else {
5         return fibonacci(n-1) + fibonacci(n-2)
6     }
7 }
```

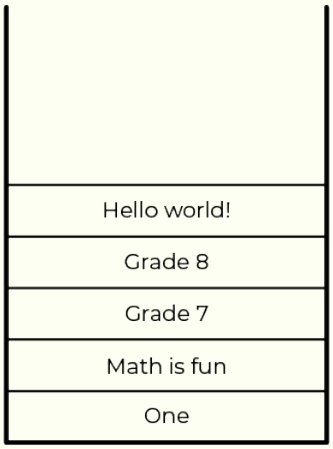
Exercise 2

Given the following pictures of the stack, fill out the table based on what would happen if you called the stack function from what the picture is showing.



empty stack

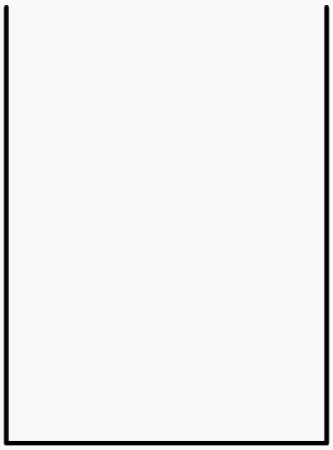
function call	return value	modifications
`stack.top()`		
`stack.is_empty()`		
`stack.is_full()`		
`stack.push(item)`		
`stack.pop()`		



non-empty stack

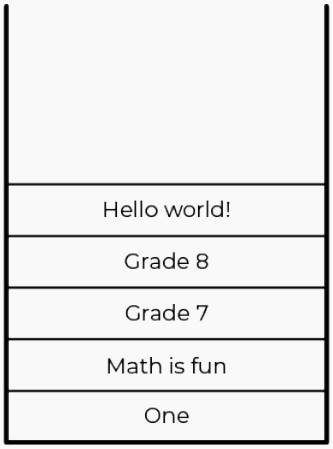
function call	return value	modifications
`stack.top()`		
`stack.is_empty()`		
`stack.is_full()`		
`stack.push(item)`		
`stack.pop()`		

Solution



empty stack

function call	return value	modifications
`stack.top()`	none	none
`stack.is_empty()`	`true`	none
`stack.is_full()`	`false`	none
`stack.push(item)`	none	puts `item` at the top of the stack
`stack.pop()`	none	none



non-empty stack

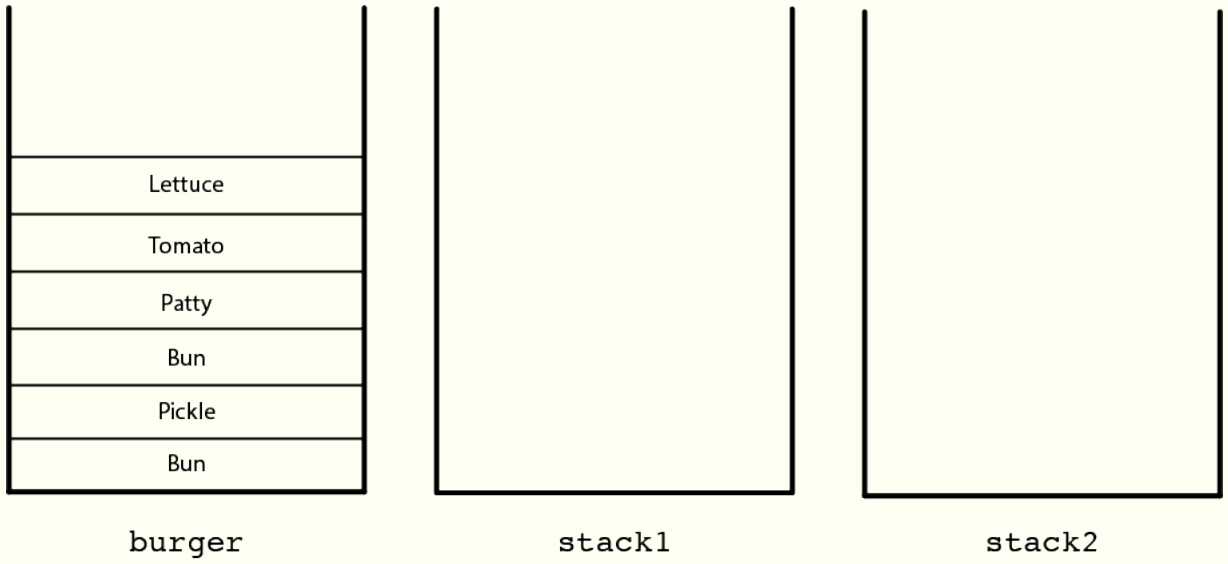
function call	return value	modifications
`stack.top()`	`"Hello World!"`	none
`stack.is_empty()`	`false`	none
`stack.is_full()`	?	none
`stack.push(item)`	none	puts `item` at the top of the stack
`stack.pop()`	`"Hello World!"`	removes `"Hello World!"` from the top of the stack

Note that:

- `top()`, `is_empty`, and `is_full` will never modify your stack.
- `push(item)` will always work.
- `pop()` can only return something if there is something on the stack.
- An item that is popped can be directly used to push onto a stack (ie. `stack.push(stack.pop())`).

Exercise 3

Suppose you had the same burger as in Example E, but instead of popping an ingredient onto the plate, you popped the ingredient to another stack. In this setup, other than your `burger` stack, you have `stack1` and `stack2`. Use stack functions (operations) to fix your burger.



Solution

Similar to Example E, we need to pop every ingredient on top of the bottom bun. But what is different is that since we are popping ingredients onto stacks, when we rebuild our burger, we can only use the ingredient on top. This means that if we pop all our ingredients onto one stack, it will be less efficient when we need the ingredient that is at the bottom of the stack. Note that there are many ways to solve this problem, some being more efficient than others.

Let's start by popping off the ingredients from `burger` and pushing what we can onto `stack1`.

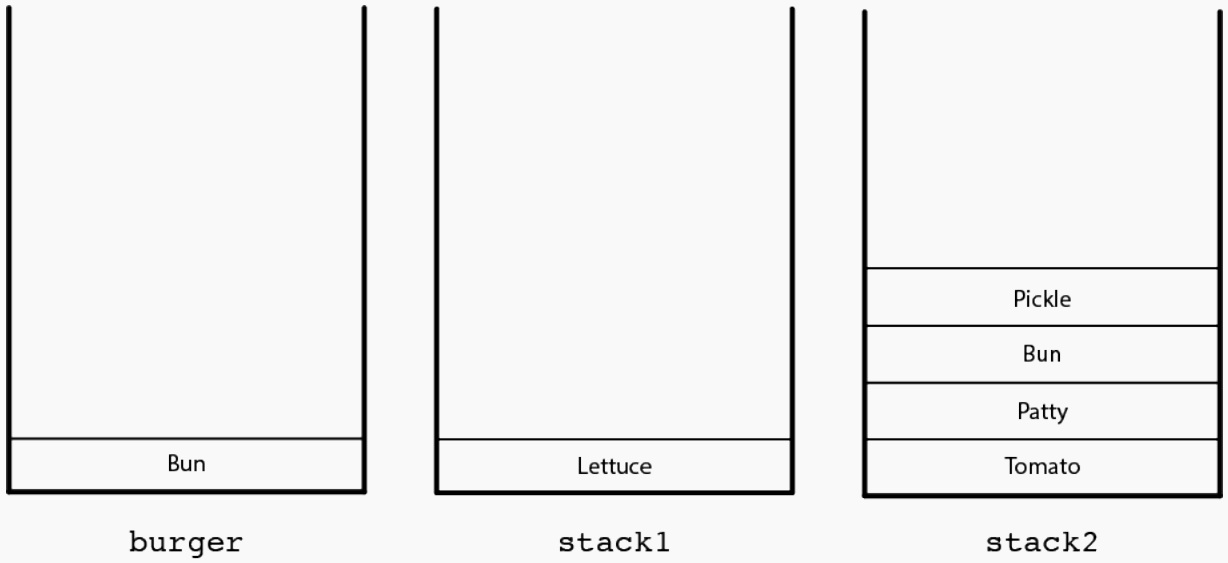
```
1  stack1.push(burger.pop())
```

This pops "Lettuce" off of `burger` and pushes it onto `stack1`. Since we want "Lettuce" to be on top of the last "Bun", we can reason that we should not push anything else onto `stack1` so that we can easily access "Lettuce" after we've popped everything we need to off

of `burger`. So we push everything else onto `stack2` instead.

```
2  stack2.push(burger.pop())
3  stack2.push(burger.pop())
4  stack2.push(burger.pop())
5  stack2.push(burger.pop())
```

Now we get the following picture:



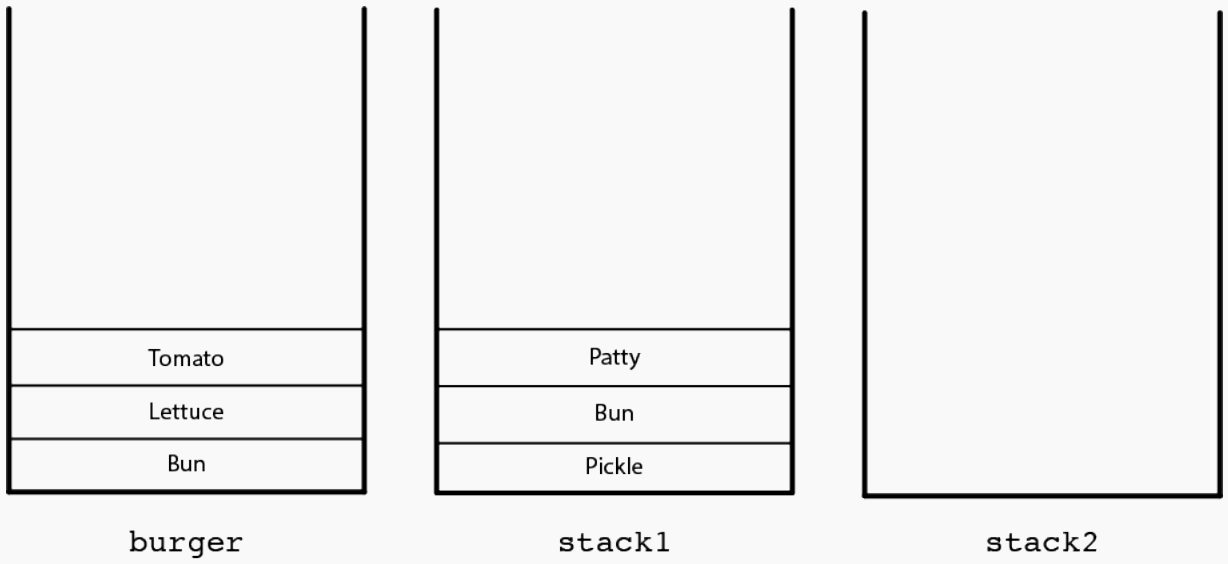
Now we start stacking the burger again. Since "Lettuce" is easily accessible, all we have to do is pop it off of `stack1` and push it onto `burger`.

```
6  burger.push(stack1.pop())   * Put the lettuce onto burger *
```

Since we need "Tomato" to be pushed onto `burger`, we need to get to the bottom of `stack2`. But we also don't want to push these other ingredients onto `burger` either, so we push it onto `stack1` since it is currently empty. So we have the following code.

```
7  stack1.push(stack2.pop())
8  stack1.push(stack2.pop())
9  stack1.push(stack2.pop())
10 burger.push(stack2.pop())   * Put the tomato onto burger *
```

Now we have the following picture:



Now we finish off our burger with the following stack operations.

```
11 stack2.push(stack1.pop())
12 stack2.push(stack1.pop())
13 burger.push(stack1.pop())   * Put the pickle onto burger *
14 stack1.push(stack2.pop())
15 burger.push(stack2.pop())   * Put the patty onto the burger *
16 burger.push(stack1.pop())   * Put the bun onto the burger *
```

Exercise 4

Write a function called `search` that takes `stack` and `item` as arguments and searches for `item` within `stack`. If the `item` is found, then return `true`. If it cannot be found, then return "Not found".

Solution

First, we want to look at the structure of our function. Our function is called `search` and have our two arguments, `stack` and `item`.

```
1 search(stack, item) {
2     if(base case condition) {
3         base case
4     } else {
5         recursive case
6     }
7 }
```

Base case: Then we look at our base case, or in other words, when is it "obvious" that we have our answer? It's "obvious" when we either have `item` at the top of the `stack` or if the `stack` is empty. So we have our following code.

```
1 search(stack, item) {
2     if(stack.is_empty()) {
3         return "Not found"
4     } else if(stack.top()=item) {
5         return true
6     } else {
7         recursive case
8     }
9 }
```

Recursive case: Lastly, we look at our recursive case, or in other words, how can we make our problem smaller.

```
1  search(stack, item) {
2      if(stack.is_empty()) {
3          return "Not found"
4      } else if(stack.top()=item) {
5          return true
6      } else {
7          stack.pop()
8          return search(stack, item)
9      }
10 }
```

Note that since `stack.pop()` modifies the `stack`, our problem gets smaller every function call.

Problem Set Solutions

1. What are the two cases you need to consider when thinking about recursion? Briefly describe each case.

Solution:
The two cases to consider are the base case and the recursive case. The base case is the smallest, most "obvious" case in the problem, and the recursive case is the when the problem is not at the base case but gets closer and closer each iteration.

2. Given the following code, identify any functions, arguments, and variables. What is the final outcome of the function call `check(main(1, 2))` on line 14?

```
1  main(arg1, arg2) {
2      return "Hello World."
3  }
4
5  check(phrase) {
6      if(phrase="Hello World!") {
7          return 1
8      } else if(phrase="Hello World.") {
9          return 2
10     } else {
11         return 3
12     }
13 }
14
15 check(main(1, 2))
```

Solution
Functions: `main`, `check`
Arguments: `arg1`, `arg2`, `phrase`
Variables: Same as arguments, since arguments are all variables.

We can trace the function call `check(main(1, 2))` as follows:

```
check(main(1, 2)) => check("Hello World.")
                  => 2
```

So the final outcome of the function call is `2`.

3. Suppose you are given a numerical grade called `grade`. A student has an A if their numerical grade is between 90-100, a B if their numerical grade is between 80-89, a C if their numerical grade is between 70-79, a D if their numerical grade is between 60-69, an E if their numerical grade is between 50-59, and F if their numerical grade is under 50. Numerical grades are only integers. Write an `if` statement that returns the letter grade.

   Hint: You can say `-1<=x<=1` to say that `x` is between -1 and 1 (inclusive).

Solution:

```
1  if(90<=grade<=100) {
2      return "A"
3  } else if(80<=grade<=89) {
4      return "B"
5  } else if(70<=grade<=79) {
6      return "C"
7  } else if(60<=grade<=69) {
8      return "D"
9  } else if(50<=grade<=59) {
10     return "E"
11 } else {
12     return "F"
13 }
```

4. Suppose we started with an empty stack named `stack` that can hold a maximum of **6 items**. Given the following code, draw a picture of the stack at the points (A), (B), and (C). Whenever `top()`, `is_empty()`, or `is_full()` is called, write the return value with its line number (ex: Write 'Line 5: "Circles"' if `stack.top()` returns "Circles").

Note that the `not` in line 18 turns `true` into `false` and `false` into `true`. In other words, it negates the value of `stack.is_full()`.

```
1  stack.is_empty()
2  stack.pop()
3  stack.push("Math")
4  stack.push("Circles")
5  stack.top()
6  * (A) *
7  stack.push("Grade 7 and 8")
8  stack.push("Recursion")
9  stack.push("and")
10 stack.push("Stack")
11 stack.push("ADTs")
12 stack.top()
13 * (B) *
14 if(stack.is_full()) {
15     stack.pop()
16     stack.pop()
17     stack.push("I love Stacks!")
18 } else if(not stack.is_full()) {
19     stack.push("I hate Stacks!")
20 } else {
21     stack.pop()
22 }
23 * (C) *
```

Solution

First, we write down all the return values of `top()`, `is_empty()`, and `is_full()`.

Line 1: `true`

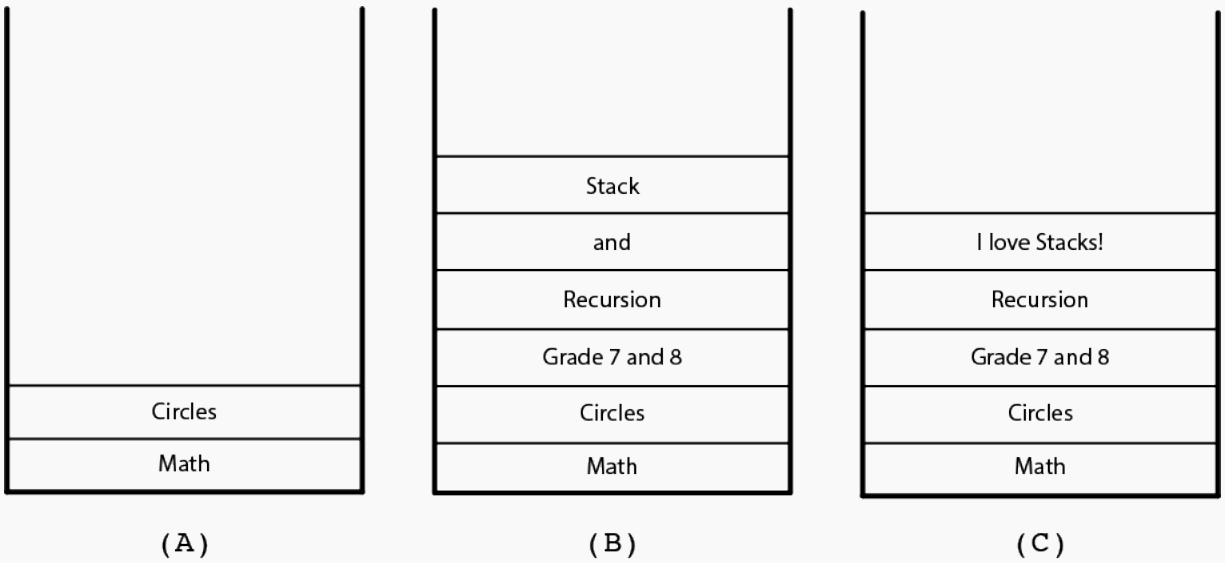
Line 5: "Circles"

Line 12: "Stack"

Line 14: `true`

Line 18: `true` (Note: the `not` turns the condition to `false`)

Then we have the following stack drawings at points (A), (B), and (C).

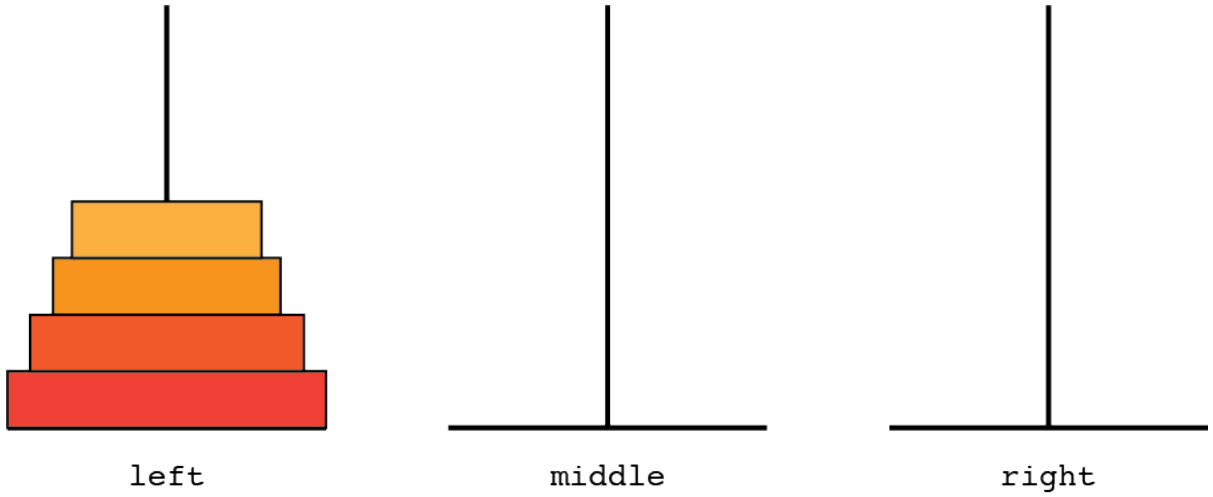


Note that "ADTs" does not end up on the `stack` in picture (B) because after we push "Stack" onto the `stack`, it has 6 items and is full! So attempting to push "ADTs" onto the `stack` will not do anything.

5. The Tower of Hanoi is a classical problem where a stack of disks is stacked biggest at the bottom to smallest at the top. There are various versions of the problem with differing amounts of disks. The goal of the problem is to move all the disks from the left stack to the right stack with the same order of disks in the end. However, one restriction is that you cannot place larger disks on top of smaller disks. Otherwise, your tower would fall! Use stack operations to solve this problem.

   We will call our stacks `left`, `middle`, and `right` respectively.

   If you would like to visualize it, you can do so here. Make sure you change the number of disks to 4. You can also try the problem with more disks if you would like. If you want to learn more about the Tower of Hanoi, you can do so here.



Solution

Since we are just moving pieces, we will only be using `push(item)` and `pop()` operations. The solution is as follows.

```
middle.push(left.pop())
right.push(left.pop())
right.push(middle.pop())
middle.push(left.pop())
left.push(right.pop())
middle.push(right.pop())
middle.push(left.pop())   * Tower of 3 disks on middle stack *
right.push(left.pop())
right.push(middle.pop())
left.push(middle.pop())
left.push(right.pop())
right.push(middle.pop())
middle.push(left.pop())
right.push(left.pop())
right.push(middle.pop())  * Tower of 4 disks on the right stack *
```

6. Suppose you are given a `stack` that you don't know the size of. Write a function called `size` that takes a `stack` and a variable `count` as arguments and returns the size of the

stack. Note that `count` will always start at 0, so we will always initially call the function like this: `size(stack, 0)`.

Solution

This is actually the same problem as the line analogy used in the introduction of the lesson!

Base case: The easiest case is to count the size of an empty stack. If the `stack` is empty, then we return the `count`. You might think that we need to return 0, but that would actually mean that we return 0 no matter the size of our initial stack. Just because our stack is empty in our base case doesn't mean that the initial stack was actually empty, because our recursive case always makes our problem smaller and smaller until we hit the base case.

Recursive case: We want to think about getting closer to our base case, which is when a stack is empty. How do we do that? The only way to make our stack more empty is to `pop` items off, so we call `stack.pop()` to do this. But we have to remember to count the item that we just popped off, so we also increase our `count` by one before recursively calling `size` again.

So this gives us the following code.

```
1 size(stack, count) {
2     if(stack.is_empty()){
3         return count
4     } else {
5         stack.pop()
6         size(stack, count+1)
7     }
8 }
```

Challenge Problem

8. Suppose you are given a `stack` and want to find an `item` and put it at the top of the stack. Suppose in addition that you are given a function called `push_to_stack` as defined below that takes two stacks, `stack` and `temp`, and pushes all the items on `temp` to `stack`. Write a function called `put_at_top` that takes two stacks, `stack` and `temp`, and a variable `item` and places `item` at the top of the stack without changing the order of the other items in the stack. If the `item` is not in the stack, then return "Error". `temp` is an empty stack, and `stack` is the stack that may or may not hold `item`.

```
1 push_to_stack(stack, temp) {
2     if(temp.is_empty()) {
3         return    * Return nothing to stop recursion *
4     } else {
5         stack.push(temp.pop())
6         push_to_stack(stack, temp)
7     }
8 }
```

Solution

Base case: There are two base cases. Either the `item` is found at the top of `stack` and we need to put it at the top of the stack or `item` is not in the function at all. The second base case happens when `stack` is empty. This is like our `search` function that we looked at in Exercise 4.

If we have the first base case, then the `item` would be at the top of the `stack`. So we need to `pop` this item off and then use `push_to_stack` to put the items that we took off `stack` and put on `temp` back onto the `stack`. Then we need to add `item` back to the stack by calling `stack.push(item)`.

Recursive case: To get closer to our base case, we need to `pop` items off of our `stack`. But since we want to keep our items in order, we need to keep our items in `temp` while we look for `item`.

So we have the following function.

```
1  put_at_top(stack, temp, item) {
2      if(stack.is_empty()) {
3          push_to_stack(stack, temp)
4          return "Error"
5      } else if(stack.top()=item) {
6          stack.pop()
7          push_to_stack(stack, temp)
8          stack.push(item)
9          return    * Return nothing to stop recursion *
10     } else {
11         temp.push(stack.pop())
12         put_at_top(stack, temp, item)
13     }
14 }
```