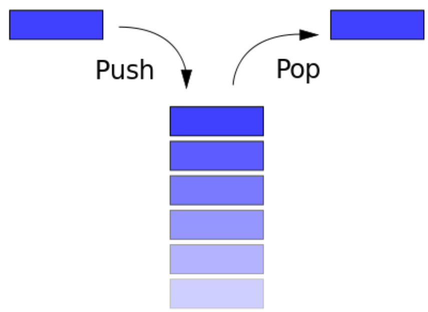
# 15-122: Principles of Imperative Computation

**Recitation 9** **Nivedita Chopra, Josh Zimmerman**

## Stacks and Queues

Stacks are a LIFO (last in, first out) data structure. This is just like what we refer to as a stack in English e.g. a stack of books.

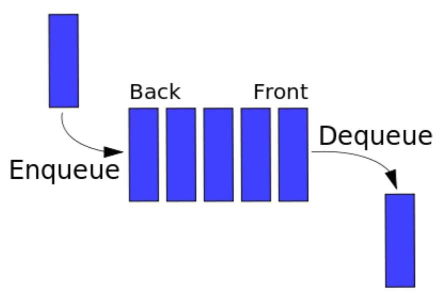



(Picture courtesy of Vegpuff/Wikipedia)

Here is the interface for stacks, as discussed in lecture:

```
stack stack_new();
bool stack_empty(stack S);
void push(stack S, string x);
string pop(stack S)
  /*@requires !stack_empty(S); @*/ ;
```

Queues are a FIFO (first in, first out) data structure. This is just like what we refer to as a queue in English e.g the queue (or line) at package pickup in the UC.




(Picture courtesy of Wikipedia)

Here is the interface for queues, as discussed in lecture:

```
queue queue_new();
bool queue_empty(queue Q);
void enq(queue Q, string x);
string deq(queue Q)
  /*@requires !queue_empty(Q); @*/ ;
```

# Checkpoint 0

Write a function to reverse a queue, using only the functions from the interface.
Below is the general structure of this function. You may not need to fill in all the blanks.

```
/* Assume that you have data types stack and queue as described in lecture */
queue reverse(queue Q)
//@requires __________
//@ensures ____________________
{
  _____________________________________ //Hint : Allocate a temporary data structure
  while(___________________________)
  //@loop_invariant ___________________
  //@loop_invariant ____________________
  {
    _____________________________________
    _________________________
  }
  ______________________________________________ //Hint : Allocate another data structure
  while(__________________________)
  //@loop_invariant ___________________
  //@loop_invariant ___________________________
  {
    _____________________________________
    ______________________________
  }
  ______________________ //Hint : Look at the function prototype
}
```

# Checkpoint 1

Why did we need those contracts in the function above?

# Checkpoint 2

Write a recursive function to count the size of a stack. At the end of the function, the stack must be unmodified (Hint : You will need to modify the stack within the function, but must ensure it is the same at the end)

## Clac

*clac* is a relatively simple *postfix*-based programming language. As we read in numbers from the input (which we represent as a queue), we push operands onto a stack and act on them based on the instructions that are in the queue.

Here's an example of *clac* processing some input (you can get this yourself when working on the clac assignment by running `clac-ref`).

```
$ clac-ref -trace
Clac top level
clac>> 5 9 2 7 3 + - / dup * %
                                 stack || queue
                                       || 5 9 2 7 3 + - / dup * %
                                     5 || 9 2 7 3 + - / dup * %
                                   5 9 || 2 7 3 + - / dup * %
                                 5 9 2 || 7 3 + - / dup * %
                               5 9 2 7 || 3 + - / dup * %
                             5 9 2 7 3 || + - / dup * %
                              5 9 2 10 || - / dup * %
                                5 9 -8 || / dup * %
                                  5 -1 || dup * %
                               5 -1 -1 || * %
                                   5 1 || %
                                     0 ||
0
```

What's happening here? Well, we push all of the numbers onto the stack after reading them out of the queue. Then, we get to the +, so we pop two items (the 7 and the 3) off of the stack, add them, and push their sum, 10, back on. Next, we get to the -, pop off the 2 and 10 and subtract them, and get -8, which we push on to the stack. Then, we get to the /. We pop 9 and -8 and divide them. 9/-8 rounds to -1, so we push that onto the stack. Next, we execute the `dup`, which simply makes the top element of the stack appear twice. We get to the `*`, which multiplies the top two elements, giving us `1`. Finally, we get to the `%`. `5 % 1 == 0`, so we push 0. Then, we're out of instructions, so we end and pop the top item off of the stack and print it.

A common source of confusion with `clac` is `if` statements and `else` statements.

When we get to an `if` statement, we pop the top item off of the stack. If it is 0, we skip the next *two* tokens in the queue – we just ignore them. Otherwise (if it's non-zero), we continue processing tokens as normal.

When we get to an `else` statement, we *always* skip the next token in the queue.

So, why are these if/else statements? Let's take a look at some `clac` code

### NOTE: In clac code below, we're using `x` it to mean any arbitrary int – you should fill in an int, like 1, -1, 0, etc, if you're actually running the code.

```
$ clac-ref -trace
Clac top level
clac>> 0 if 2 else 3
                                 stack || queue
                                       || 0 if 2 else 3
                                     0 || if 2 else 3
                                       || 3
                                     3 ||
3
clac>> 1 if 2 else 3
                                 stack || queue
                                     3 || 1 if 2 else 3
                                   3 1 || if 2 else 3
                                     3 || 2 else 3
                                   3 2 || else 3
                                   3 2 ||
2
```

Next, let's write a simple clac program: one that calculates absolute value. We can define $|x|$ as follows:

$$|x| = \begin{cases} x * 1 & \text{if } x \geq 0 \\ x * -1 & \text{if } x < 0 \end{cases}$$

So, if $x$ is less than 0, we want to multiply it by $-1$ and otherwise we want to multiply it by $1$. If we run the clac command `x 0 <`, then it will result in 1 being on the top of the stack if $x < 0$ and 0 being on the top of the stack otherwise.

We eventually want to multiply by either 1 or -1, so we should push the appropriate one of them onto the stack: If $x < 0$ we multiply by -1, otherwise we multiply by $1$.

So, we add `if -1 else 1` to our command. Now we have

```
x 0 < if -1 else 1
```

This says "if $x < 0$, push -1 onto the stack. Otherwise, push 1 onto the stack." This works because when `x 0 <` evaluates to 0 (so $x \geq 0$), we ignore the tokens `-1` and `else`, so we just push 1 onto the stack. If `x 0 <` evaluates to 1 (so $x < 0$), then we push -1 onto the stack and ignore the token 1.

Next, we want to multiply by x, so we add * to the end:

```
x 0 < if -1 else 1 *
```

This doesn't work, though! We popped x off of the stack when we did the comparison. If we run the above command, we get:

```
Error: Error: not enough elements on stack
```

So, we need to duplicate x before we compare, so we can still use it later:

```
x dup 0 < if -1 else 1 *
```

That will compute the absolute value of x.