15-122: Principles of Imperative Computation

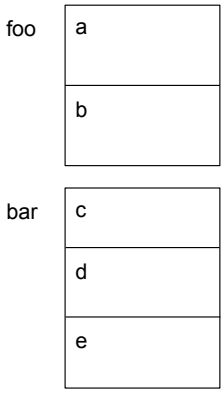
Recitation 4: A `queue_t` Interface

Nivedita, Andrew, Aaron

A Wild `struct` Appears

Suppose we have the following definitions:

```
1  struct X {
2    int a;
3    struct Y* b;
4  };
5
6  struct Y {
7    int* c;
8    int d;
9    struct X* e;
10 };
11
12 struct X* foo = alloc(struct X);
13 struct Y* bar = alloc(struct Y);
14
15 foo->b = bar;
16 bar->e = foo;
17
18 bar->e->a = 15;
19 foo->b->c = alloc(int);
20 *(bar->c) = foo->a * 8 + 2;
21 foo->b->d = 1000 * foo->a + *(foo->b->c);
```



Checkpoint 0

Fill out the table above. What's the value of `bar->d`? (For your own sanity, draw a picture!)

Stack and Queue Interfaces

In lecture we discussed four functions exposed by the stack interface:

- stack_new: Creates and returns a new stack
- stack_empty: Given a stack, returns true if it is empty, else false
- push: Given a stack and a string, puts the string on the top of the stack
- pop: Given a stack, removes and returns the string on the top of the stack

Similarly, we discussed four functions exposed by the queue interface:

- queue_new: Creates and returns a new queue
- queue_empty: Given a queue, returns true if it is empty, else false
- enq: Given a queue and a string, puts the string at the end of the queue
- deq: Given a queue, removes and returns the string at the beginning of the queue

Checkpoint 1

Write a function to reverse a queue using only functions from the stack and queue interfaces.

```
1  void reverse(queue_t Q) {
2
3      _____________________________________________ // Hint : Allocate a
4                                                    // temporary data structure
5      while(_____________________________________) {
6
7          ___________________________________________
8
9          ___________________________________________
10     }
11     while(_____________________________________) {
12
13         ___________________________________________
14
15         ___________________________________________
16     }
17 }
```

Checkpoint 2

Write a recursive function to count the size of a stack. You may not destroy the stack in the process - the stack's elements (and order) must be the same before and after calling this function.

```
1  int size(stack_t S) {
2
3      ___________________________________________________________
4
5      ___________________________________________________________
6
7      ___________________________________________________________
8
9      ___________________________________________________________
10
11     ___________________________________________________________
12
13     ___________________________________________________________
14 }
```

Checkpoint 3

Why couldn't this stack size implementation be used in contracts in C0? Hint: Contracts in C0 cannot have side effects.