1. Describe how to implement a quack with ***three*** black-box stacks, so that each quack operation requires $O(1)$ amortized stack operations.

Solution (clones and ghosts; partial credit): Let's call the three stacks *QIn*, *QOut*, and *Stack*. The first two stacks *QIn* and *QOut* will simulate a queue; the third stack *Stack* will behave like a normal stack.

Any item that is really in the quack appears exactly twice in our data structure: once in either *QIn* or *QOut*, and once in *Stack*. However, our structure may also contain at most one *ghost* copy of each item that the user has already extracted from the quack, either at the bottom of *QOut* or at the bottom of *Stack*. We also keep track of the number of ghosts in these two stacks. Whenever we discover that every item on a stack is a ghost, we empty that stack.

```
QUACKPUSH($qck, x$):
  PUSH($qck.QIn, x$)
  PUSH($qck.Stack, x$)
```

```
QUACKPOP($qck$):
  if $qck.QIn$ is not empty
      POP($qck.QIn$)
  else
      HAUNT($qck.QOut$)
  return POP($qck.Stack$)
```

```
QUACKPULL($qck$):
  if $qck.QOut$ is empty
      while $qck.QIn$ is not empty
          PUSH($qck.QOut$, POP($qck.QIn$))
  HAUNT($qck.Stack$)
  return POP($qck.QOut$)
```

```
⟨⟨Add a ghost to stack S; clear if all ghosts⟩⟩
HAUNT($S$):
  $S.ghosts \leftarrow S.ghosts + 1$
  if $S.ghosts = S.num$
      while $S$ is not empty
          POP($S$)
      $S.ghosts = 0$
```

Consider a sequence of N quack operations. In the worst case, a single quack operation could require $O(N)$ stack operations. However, each item inserted into the quack is pushed and popped from each of the three stacks at most once during its lifetime. Thus, any sequence of N quack operations leads to at most $6N$ stack operations, which implies that each quack operation uses $O(1)$ amortized stack operations.

One *significant* downside of this solution is that its memory footprint is not bounded by *any* function of the number of items in the quack. For example, suppose we perform N PUSHES and then $N-1$ PULLS on an initially empty quack. The resulting abstract quack contains only one item, but in our implementation, *Stack* still contains $N-1$ ghosts, and therefore uses $\Omega(N)$ space. ■

Solution (no clones; no ghosts; full credit): I will implement the quack as a triple of black-box stacks *Top*, *Bot*, and *Tmp*, and an integer counter *num*, which stores the number of items in the quack. After each quack operation, stack *Tmp* is empty and each item in the quack is stored in exactly one of the other two component stacks.

Intuitively, pushes insert items into *Top*, pops remove items from *Top*, and pulls remove items from *Bot*. However, whenever we need to pop from an empty stack, we first transfer *half* of the items from the other stack, using *Tmp* to maintain the correct item order.

```
InitQuack():
    $qck \gets$ new quack
    $qck.num \gets 0$
    $qck.Top \gets$ InitStack()
    $qck.Bot \gets$ InitStack()
    $qck.Tmp \gets$ InitStack()
    return $qck$
```

```
QuackIsEmpty(qck):
    return $qck.num = 0$
```

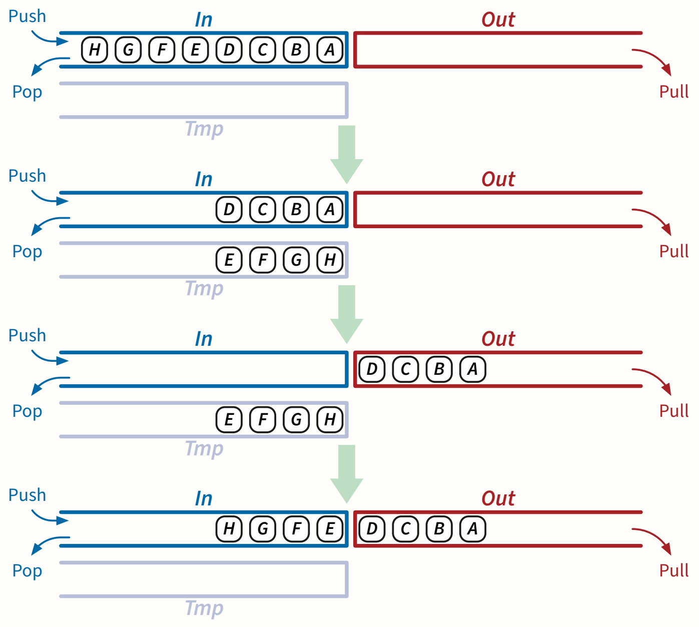
```
QuackPush(qck, x):
    Push($qck.Top, x$)
    $qck.num \gets qck.num + 1$
```

```
QuackPop(qck):
    if IsEmpty($qck.Top$)
        QuackBalance($qck, qck.Bot, qck.Top$)
    $qck.num \gets qck.num - 1$
    return Pop($qck.Top$)
```

```
QuackPull(qck):
    if IsEmpty($qck.Bot$)
        QuackBalance($qck, qck.Top, qck.Bot$)
    $qck.num \gets qck.num - 1$
    return Pop($qck.Bot$)
```

```
⟨⟨Move the deep half of stack Src to stack Dst⟩⟩
QuackBalance(qck, Src, Dst):
    for $i \gets 1$ to $\lceil qck.num/2 \rceil$
        Push($qck.Tmp$, Pop($Src$))
    while ¬IsEmpty($Src$)
        Push($Dst$, Pop($Src$))
    while ¬IsEmpty($qck.Tmp$)
        Push($Dst$, Pop($qck.Tmp$))
```

The only part of these algorithms that does not run in $O(1)$ worst-case time is QuackBalance, runs in $O(n)$ time. The following figure shows an example of QuackBalance in action.



Consider the sequence of quack operations between two consecutive QuackBalance calls. Let n_0 and n_1 denote the values of *qck.num* just after the first QuackBalance and just before the second. To simplify the case analysis, I'll assume that n_0 is even; after the first QuackBalance, stacks *In* and *Out* each contain exactly $n_0/2$ items. (Dealing with odd n_0 only changes the constants in $O(1)$ amortized time bounds.)

- Just before the second QuackBalance, one of the stacks *In* or *Out* is empty, so we must have performed at least $n_0/2$ deletions (QuackPops and QuackPulls). Thus, if $n_0 \geq n_1$, we can charge the $O(n_1) = O(n_0)$ stack operations for the second rebalance to these $\Omega(n_0)$ deletions.
- On the other hand, if $n_1 > n_0$, then just before the second QuackBalance, at least half of the n_1 items in the non-empty stack *Top* were QueuePushed after the first rebalance. Thus, we can charge the $O(n_1)$ stack operations for the second QuackBalance to these $\Omega(n_1)$ QueuePushes.

In both cases, $O(1)$ stack operations are charged to each quack operation.

Assuming the component stacks are space-efficient, this data structure always uses $O(n)$ space to store n items, unlike the cones-and-ghosts data structure. ■

2. (a) Describe how to implement a steque with ***two*** black-box stacks, so that each steque operation requires $O(1)$ amortized stack operations.

Solution: We make only minor modifications to our implementation of queues with two stacks. I will implement the steque as a pair of black-box stacks *In* and *Out*, and an integer counter *num*, which stores the number of items in the steque. After each steque operation, each item in the steque is stored in exactly one of the two component stacks.

Intuitively, items are shoved into *In*, pushed into *Out*, and popped from *Out*. But if *Out* is empty when we try to pop, we first transfer all items from *In* to *Out*, similarly to our queue simulation. If there are no pushes, this *is* our queue implementation, but using shove and pop instead of push and pull, and swapping the roles of *In* and *Out*.

```
InitSteque():
    $stq \leftarrow$ new steque
    $stq.num \leftarrow 0$
    $stq.In \leftarrow$ InitStack()
    $stq.Out \leftarrow$ InitStack()
    return $stq$
```

```
StequeIsEmpty($stq$):
    return $stq.num = 0$
```

```
StequePush($stq, x$):
    Push($stq.Out, x$)
    $stq.num \leftarrow stq.num + 1$
```

```
StequeShove($stq, x$):
    Push($stq.In, x$)
    $stq.num \leftarrow stq.num + 1$
```

```
StequePop($stq$):
    if IsEmpty($stq.In$)
        ⟨⟨Transfer items from In to Out⟩⟩
        while ¬IsEmpty($stq.In$)
            Push($stq.Out$, Pop($stq.In$))
    $x \leftarrow$ Pop($stq.In$)
    $stq.num \leftarrow stq.num - 1$
    return $x$
```

The only part of these algorithms that does not run in $O(1)$ worst-case time is the *transfer* loop in StequePop, which moves the contents of stack *In* to stack *Out* in $\Theta(n)$ time.

Consider the sequence of operations between two consecutive transfers. Let n_0 and n_1 respectively denote the values of *stq.num* just after the first transfer and just before the second. Between these two transfers we must perform *exactly* n_1 StequeShoves (and *at least* n_0 StequePushes). Thus, we can charge the $O(n_1)$ stack operations in the second transfer to these n_1 StequeShoves. Each StequeShove is charged for $O(1)$ stack operations; StequePushes and StequePops are not charged at all.

Alternatively, each item that is ever inserted into the steque participates in at most four stack operations. There are only two cases to consider:

- Inserted by QuackShove: shoved into *In*, pulled from *In*, pushed into *Out*, and popped from *Out*.
- Inserted by QuackPush: pushed into *Out*, and popped from *Out*.

Thus, each QuackShove uses 4 amortized stack operations, QuackPush uses 2 amortized stack operations, and QuackPop uses ***zero*** stack operations.[a] ■

[a]We can't use this global argument for our space-efficient quack, because a single item can be involved in arbitrarily many calls to QuackBalance.

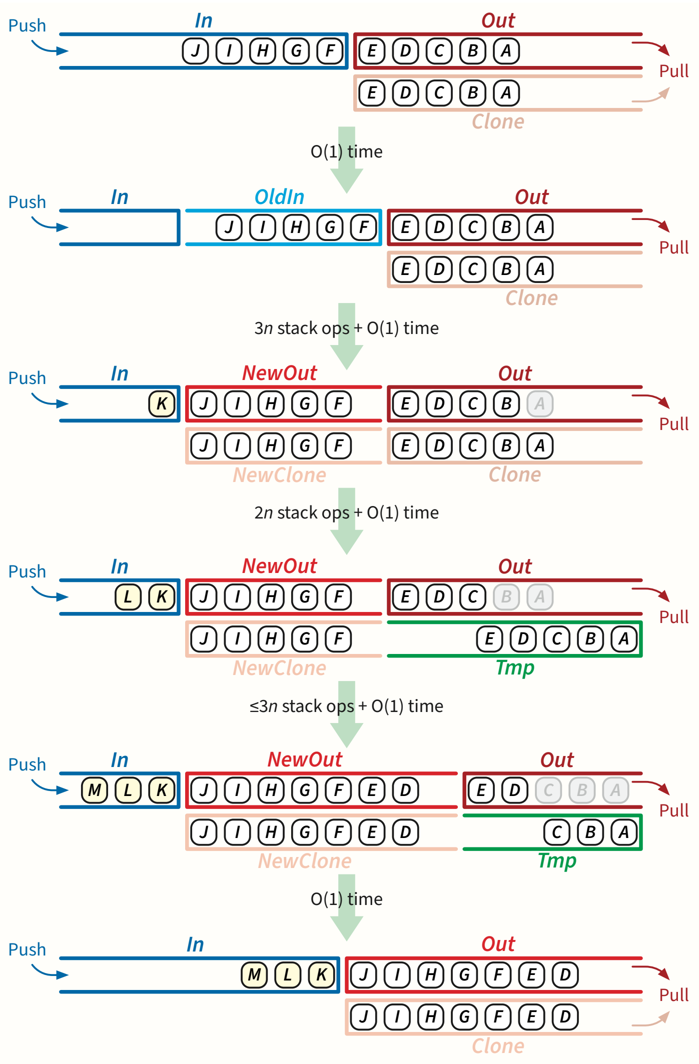
4. Describe how to implement a queue with $O(1)$ black-box stacks, so that each queue operation takes $O(1)$ stack operations *in the worst case*.

Solution: I will implement the queue using ***seven*** stacks: *In*, *Out*, *Clone*, *OldIn*, *Tmp*, *NewOut*, and *NewClone*. Each item in the queue is stored once in either *In* or *Out*, but copies of that item may also be stored in other stacks. We also maintain counters $S.num$ storing the number of items in each stack S. (I'll pretend these counters are part of the black-box stacks, but we can maintain them separately if necessary.)

As long as $In.num < Out.num$ (or the queue is empty), the stacks *In* and *Out* behave exactly like their counterparts in the amortized queue implementation, and *Clone* is an exact copy of *Out*. However, whenever $In.num = Out.num$, we start transferring all items from *In* to *Out*, using the other stacks either as temporary storage or as backups. Here is a description of the transfer process as a standalone algorithm. (To simplify the algorithm, I'm assuming that our dynamic memory model includes garbage collection. Explicitly emptying and deleting stacks when they become useless only increases the final constant running time.)

```
TRANSFER():
    ⟨⟨—— Initialize transfer stacks ——⟩⟩
    $OldIn \gets In$
    $In \gets$ new stack
    $NewOut \gets$ new stack
    $NewClone \gets$ new stack
    ⟨⟨—— Transfer OldIn to NewOut and NewClone ——⟩⟩
    while $OldIn$ is not empty
        $x \gets OldIn.\text{POP}()$
        $NewOut.\text{PUSH}(x)$
        $NewClone.\text{PUSH}(x)$
    ⟨⟨—— Transfer Clone to NewOut and NewClone ——⟩⟩
    $Tmp \gets$ new stack
    while $Clone$ is not empty
        $Tmp.\text{PUSH}(Clone.\text{POP}())$
    $fromIn \gets NewOut.num$
    while $NewOut.num < fromIn + Out.num$
        $x \gets Tmp.\text{POP}()$
        $NewOut.\text{PUSH}(x)$
        $NewClone.\text{PUSH}(x)$
    ⟨⟨—— Reset to normal behavior ——⟩⟩
    $Out \gets NewOut$
    $Clone \gets NewClone$
    $xfer \gets$ FALSE
```

This TRANSFER process is illustrated on the next page.



In fact, we spread the execution of Transfer across the next several queue operations after *In.num* = *Out.num*, as described below. The subroutine TransferStep() performs the next 12 stack operations in Transfer, plus $O(1)$ additional work. (That is, one call to TransferStep() either transfers four items from *In*, six items from *Clone*, or four items from *Tmp*.) While Transfer is active (indicated by the boolean flag *xfer* being True), we Push new items into stack *In* as usual, and we Pull items from *Out* as usual, but Pull does not directly modify *Clone*.

```
QueuePush(x):
  if xfer
      TransferStep()
  In.Push(x)
  if In.num ≥ Out.num
      xfer ← True
```

```
QueuePull():
  if xfer
      TransferStep()
  else
      Clone.Pop()
  x ← Out.Pop
  if In.num ≥ Out.num
      xfer ← True
  return x
```

If $n = \mathit{In.num} = \mathit{Out.num}$ when TRANSFER begins, then TRANSFER performs at most $3n+2n+3n = 8n$ stack pushes and pops, plus $O(1)$ other work. Thus, TRANSFER is complete after $2n/3$ queue operations. This implies that TRANSFER is complete before the old *Out* stack becomes empty, and that after TRANSFER, we have more items in *Out* than *In*.

This solution is adapted from a description by Hood and Melville [1], which uses only *six* stacks. The difference between their six-stack solution and my seven-stack solution is that Hood and Melville's stacks are standard LISP lists, not black boxes. Implementing stacks in a functional language like LISP automatically makes them *fully persistent*, meaning old versions of stacks are still available to be queried or even modified. In Hood and Melville's implementation, "*Clone*" begins as the version of "*Old*" when TRANSFER begins. In `git` terminology, *Clone* is a branch of *Out*. In imperative terms, Hood and Melville's stacks are implemented as linked lists, except that *Out* and *Clone* are pointers into the same linked list.

In short, they cheated.[a]

Okasaki [2] describes several purely functional queues that are more efficient than Hood-Melville. I don't know whether these can be adapted to fast queue implementations with fewer black-box stacks.

[1] Robert T. Hood and Robert V. Melville. Real time queue operations in pure LISP. *Information Processing Letters* 13(2):50-54, 1981. Technical Report TR80-422, Department of Computer Science, Cornell University, 1980.

[2] Chris Okasaki. Simple and efficient purely functional queues and deques. *Journal of Functional Programming* 5(4):583–592, 1995. See also Chapter 5 in *Purely Functional Data Structures*.

■

[a] More accurately, they played a different and more interesting game than the one I'm playing.