1 Question 1

1.1 Part (a)

To implement DECREMENT:

```
function DECREMENT(A, k)
    i ← 0
    while i < k ∧ A_i = 0 do
        A_i ← 1
        i ← i + 1
    end while
    if i < k then
        A_i ← 0
    end if
end function
```

To achieve an $\Omega(k)$ average operation cost, on a k-bit counter that initially contains x ($0 \leq x < 2^k$), supposing that each change of a value in A has unit cost, there are 2 cases:

- $x = 0$: In this case, do a single DECREMENT operation. The cost for this operation is k (as it will set all k bits of the counter to 1), and thus, the average cost for the operation is also k i.e. is $\Omega(k)$.
- $x > 0$: First do x DECREMENT operations (setting the counter to 0), and then alternate DECREMENT and INCREMENT x times. The first x operations cost at least 1 each, whereas the next x operations (which consist either of DECREMENT-ing a counter that contains 0, or INCREMENT-ing a counter that contains $2^k - 1$) each have cost k. Thus the average cost per operation is at least $\frac{x+xk}{2x} = \frac{k+1}{2}$ i.e. is $\Omega(k)$.

1.2 Part (b)

To efficiently implement RESET, while using only k extra bits, maintain another array of bits $B[0..k)$ that satisfies the following invariant:

$$B_i = A_i \vee A_{i+1} \vee ... \vee A_{k-1}$$

B will thus initially contain only 0's. Now to see the effects of the various operations. An INCREMENT operation will set a bit in B to 1 whenever a corresponding bit in A is set to 1. Also, when an INCREMENT overflows the counter, B is set entirely to 0:

function INCREMENT(A, B, k)
    $i \leftarrow 0$
    while $i < k \land A_i = 1$ **do**
        $A_i \leftarrow 0$
        $i \leftarrow i + 1$
    end while
    if $i < k$ **then**
        $A_i \leftarrow 1$
        $B_i \leftarrow 1$
    else
        while $i \geq 0$ **do**
            $B_i \leftarrow 0$
            $i \leftarrow i - 1$
        end while
    end if
end function

To implement RESET, use the array B to decide ”how far” to go in A when setting bits to 0. This works since, once B becomes 0, by the invariant, A will have no further bits to reset:

function RESET(A, B, k)
    $i \leftarrow 0$
    while $i < k \land B_i = 1$ **do**
        $A_i \leftarrow 0$
        $B_i \leftarrow 0$
        $i \leftarrow i + 1$
    end while
end function

To show that this has amortized $O(1)$ complexity, assuming that each operation that sets a value in A or B has unit cost, consider the following potential function on the state of the data structure:

$$\Phi(A, B, k) := f_1(A) + 2f_1(B)$$

Where $f_1(X)$ represents the number of times the bit 1 appears in an array X. This is a valid potential function since it is initially zero, and is by definition always non-negative and real.

I now analyse each operation's cost separately:

- INCREMENT: Here there are two cases:
  - If no overflow occurs, suppose that x bits are set to 0 in lines 3-6. Since no overflow occurs, we enter the **if** on lines 7-10, not the **else** on lines 10-14. Thus, we do $x+2$ actual operations; however, the potential is reduced by x by lines 3-6, and increased by 3 on lines 7-10. We thus have an amortized cost of $x+2-x+3=5$ i.e. $O(1)$.
  - If overflow occurs, then lines 3-6 will set k bits to 0. Since overflow occurs, we enter the **else** on lines 10-14, not the **if** on lines 7-10. Thus we do a further k operations in setting all bits of B to 0. However, the potential is reduced by k by lines 3-6 and by a further $2k$ by lines 10-14. Thus the amortized cost of the operation in this case is $k+k-k-2k=-k \leq 0$ i.e. $O(1)$.
- RESET: Here, suppose the **while** on lines 3-6 runs x times. We thus do $2x$ actual operations. On the other hand, each iteration will certainly set a bit in B from 1 to 0, reducing the potential by 2. While we may also reduce the potential by setting a bit in A from 1 to 0, this will not necessarily happen on every iteration. Thus, the potential is reduced by at least $2x$, leading us to an amortized cost of at most $2x-2x=0$ i.e. $O(1)$.

So, in all cases, we have an $O(1)$ amortized cost per operation.

2 Question 2

To implement such a data structure, allocate two stacks A and B that support operations POP, PUSH and EMPTY, where the POP operation not only removes the top element from a stack, but also returns it. I assume these to take unit time. I also assume that DEQUEUE is required not only to remove an element from the queue, but also return it. Thus, implement ENQUEUE and DEQUEUE as follows:

```
function ENQUEUE(A, B, x)
    PUSH(A, x)
end function
function DEQUEUE(A, B, x)
    if EMPTY(B) then
        while ¬ EMPTY(A) do
```

7: $\quad\quad$ PUSH(B, POP(A))
8: $\quad$ **end while**
9: **end if**
10: **return** POP(B)
11: **end function**

To see why this is correct, note that if our data structure represents a queue which holds a sequence S (where we ENQUEUE to the front of S and DEQUEUE from the back of S), then $S = \langle A \rangle \| reverse\langle B \rangle$, where $\langle X \rangle$ represents the contents of some stack X taken from top to bottom, and $\|$ represents concatenation. With this in mind, note that ENQUEUE inserts x to the top of A, thus indeed putting x at the beginning of S. DEQUEUE may initially, if B is empty, set B to $reverse(A)$ and erase all elements from A (which overall does not change S, as S is initially $\langle A \rangle \| \epsilon$, and after this it is $\epsilon \| reverse(reverse\langle A \rangle)$, where ϵ is the empty sequence – and these two sequences are equal) after which it returns the element at the top of B, which indeed represents the last element of S.

To show that this data structure has $O(1)$ amortized complexity for either operation, consider the following potential function:

$\Phi(A, B) := |A|$

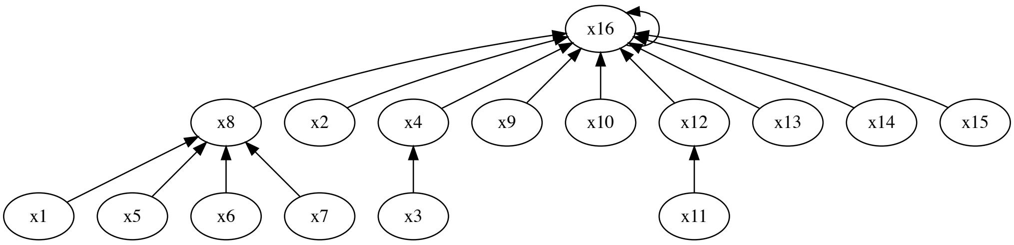
Where $|A|$ is the size of A.

I now show that each operation in turn has $O(1)$ amortized complexity:

- ENQUEUE: Here we do 1 actual operation, and increase $\Phi(A, B)$ by 1. So the amortized cost is 2 i.e. $O(1)$.
- DEQUEUE: There are 2 cases:
  - if the **if** on line 5 is triggered, we do $|A|$ operations in the **while** on lines 6-8. Additionally we do 1 further operation on line 10. On the other hand, the **while** also reduces the potential by $|A|$, since it empties stack A, which leads to an amortized cost of $|A| + 1 - |A| = 1$ i.e. $O(1)$.
  - otherwise, we only do the 1 operation on line 10, and we do not modify the potential, so the amortised cost is $O(1)$.

So overall the data structure has $O(1)$ amortised time complexity.

3 Question 3



4 Question 4

Note that:
$log(n) = log(65536^{65536^{65536}}) = log(2^{16*65536^{65536}}) = 16 * 65536^{65536}$
$log(log(n)) = log(16 * 65536^{65536}) = 4 + log(2^{4*65536}) = 4 + 4 * 65536$
$log(log(log(n))) = log(4 + 4 * 65536) \approx 18.00002$
$log(log(log(log(n)))) \approx log(18.00002) \approx 4.169$
$log(log(log(log(log(n))))) \approx 2.059$
$log(log(log(log(log(log(n)))))) \approx 1.042$
$log(log(log(log(log(log(log(n))))))) \approx 0.059$
This implies that $log^*(n) = 7$.

5 Question 5

To show this, take $n = 2^k$, for any $k \in \mathbb{N}$, and let m be equal to $2n - 1$. Now use the following operations on initial values $1...n$:

for $i \leftarrow 1...k$ **do**
  for $j \leftarrow 2^{i-1}...n - 2^{i-1}$ **step** 2^i **do**
    UNION$(j, j + 2^{i-1})$
  end for
    ▷ after this, nodes $2^i, 2 * 2^i, 3 * 2^i, ..., n$ have subtree height i
end for
▷ Now, n has subtree height $k = log_2(n)$, and we have done $n - 1$ operations so far. Note also, that at this point, 1 is a leaf at depth $k = log_2(n)$

for $i \leftarrow 1 ... n$ **do**
  FIND(1)
end for

Now note that:

- There are $n-1$ UNION's, but, for the purpose of this analysis, we can ignore their cost. This is valid as we are asked to prove a lower bound on the total cost.
- We use n FIND operations, and each takes time proportional to the depth of node 1 i.e. $log_2(n)$. Since, again, $m = 2n-1$, this implies that we use $\Omega(m\ log_2(n))$ time for these operations.

This means that we use $\Omega(m\ log(n))$ time overall.