1. We need to write the methods `enqueue()` and `dequeue()`. We will assume that two stacks named `in_stack` and `out_stack` are created by the constructor of the queue class.

```python
def enqueue(a):
    in_stack.push(a)

def dequeue():
    if out_stack is not empty:
        return out_stack.pop()
    elif (out_stack is empty) and (in_stack is not empty):
        while in_stack is not empty:
            t = in_stack.pop()
            out_stack.push(t)
        return out_stack.pop()
    else:
        return empty_queue_exception
```

We can easily see that worst case `enqueue()` is $O(1)$ and `dequeue()` is $O(n)$. To show that `dequeue()` is amortized $O(1)$ we focus on the case that causes all the work, i.e., the while-loop, as all other parts of `dequeue` run in worst case $O(1)$. In this loop there are $n$ size checks, $n$ pops, and $n$ pushes. If we think of saving three cyber dollars for every enqueue, placing them on the item being enqueue. Then when it comes time to move the item from the `in_stack` to the `out_stack` this move is already payed for. Since we have only added $O(1)$ to `enqueue` it is still $O(1)$, and not `dequeue` is also $O(1)$, as the $O(n)$ while loop is now free.

2. Preform a depth first traversal of the tree storing the depths of each node in a list. Then sum the values in the list. The time for the traversal is $O(n)$, and the space is $O(n)$. In addition summing the list takes $O(n)$ time. Hence, the algorithm runs in $O(n)$ time and space.

3. We layout the track as in Figure 1. With this layout we can insert and remove trains from either the front or rear of the storage track. Furthermore, we can not modify the order of the trains in the storage track in any other way.

![Figure 1: Train track representation of a dequeue.](image-url)
4. **Linked-list**: In the linked list version we need $O(k)$ to find the next person to be removed in every iteration. The removal itself is $O(1)$ work. Hence, the total runtime is $O(nk)$, as there are $n$ iterations.

**Array**: In the array version we need only $O(1)$ work to find the next person, as we can just compute $(a + k) \mod l$ where $a$ is the current position and $l$ is the current length of the array. However, after every deletion we need to shift the array and close the gap which is $O(n)$ work. Hence the worst case runtime is $O(n^2)$.

5. First we compute the depth of each employee in the tree. This can be done in $O(h)$ time by walking to the root (following parent pointers), and counting the number of nodes above the employee. Now we can implement a recursive solution which runs in $O(d)$ time where $d$ is the distance between the two nodes in the tree, making the total runtime $O(h)$.

```python
def lca(a, depth_a, b, depth_b):
    if a == b:
        return a
    if depth_a <= depth_b:
        return lca(a.parent, depth_a - 1, b, depth_b)
    else:
        return lca(a, depth_a, b.parent, depth_b - 1)
```