1. Queue can be implemented using two stacks (a and b) as follows:

   Enqueue operations are done using stack a – as and when an element is enqueued, it is added on top of stack a.
   Dequeue operation requires both stacks in this design – pop each element form stack a, and enqueue it to stack a. As we index into the last element, we can pop it and dequeue it successfully.

   Each of the enqueue operation runs in O(1) time. The total cost of performing n enqueue operation is O(n) time in worst-case. The average cost, or amortized cost, per operation can be defined as O(n)/n which is approximated as O(1), constant runtime.

   Each of the dequeue operation involves multipop (sequential popping of multiple elements).

   Pseudocode for multipop operation:

   ```
   While elements still exist in the stack:
       Call the pop operation
   ```

   Owing to these iterations, the multipop function runtime is linear, dependent on the number of elements in the stack that we want to pop out (in the case of dequeue, it will be all the elements that exist in the stack).

   The number of times an object can be popped is as many times it is pushed onto the stack, hence the runtime for this function is O(1). If this function is called n times, the runtime for n calls is O(n) – the amortized cost per operation is then O(n)/n = O(1).

   The above analysis is based on the principles of The Aggregate Method of Amortized Analysis.

2. The runtime analysis for deques are as follows:

Using the Accounting Method for Amortized Analysis, I will assign the following amortized costs before I begin to analyze the functions:

PUSH: 2
POP: 0
MULTIPOP: 0

This is in accordance with the rule that while we push an element onto the stack, we have already charged the amortized time that would be required to both push and pop this element to/from the stack. Hence, we have charged enough amortized time to pop each of the elements that we have pushed to the stack, which means that we have charged at least the amount of amortized time required for multipop. Using this system, the total amortized cost of the functions would be O(n), with an average amortized cost as O(1).

Push functions for stacks have total amortized time as O(n) for n elements being pushed, and an average amortized cost as O(1). This is a simplification, since the runtime depends on if stacks a and b are empty or not. If either of these are empty, the runtime is O(2*n), which is simplified as O(n).

Pops are being ‘paid’ for in the pushes, so if this method for analysis holds true, the pops would effectively take no considerable amount of runtime. One of the pop cases is when either a or b is empty. That involves the multipop function, which is still being ‘paid’ for in the pushes. Hence, the combined average for pushes and pulls can be stated as amortized O(1).

3. I will use the Potential Method for Amortized analysis of this data structure. Using the standard definition of the potential function, I will define it on a queue to be the number of objects in the queue. For the empty list, the potential function value is 0. The potential is always nonnegative, since the stack is never negative. To depict the potential function, I will use f() notation.

Enqueue operation: For inserting an element in the beginning of the queue, the change in potential is as follows:
Let the initial length of the queue be l
Change in potential: f(l + 1) – f(l) = (l + 1) – (l) = 1
Hence, the amortized cost for this operation = time cost + 1 = 2 (Using the standard potential function equation)

Dequeue operation: For popping an element from the end of the queue, the change in potential is as follows:
Let the initial length of the queue be m
Change in potential: f(m-1) – f(m) = (m-1)-(m) = -1
Hence, the amortized cost for this operation = time cost – 1 = 0 (Using the standard potential function equation)

Min operation: For computing the minimum of the queue, I have used the Quick sort function.
The worst-case runtime would be $O(n^2)$, but the general case runtime can be approximated as O(n.log(n)). The change in potential is (a) pushing of all the elements from the queue to a list data structure, and (b) performing multiple pops and pushes until the list is sorted (which is net 0 amortized cost). The average cost for this operation would be O(log(n)).

For proving the correctness of the functions, I have included some tests in the python file.