Stack's push operation using enqueue and dequeue:

```
public void push(T item) throws Exception {
        // First detects if the stack is full
        if (isFullStack()) {
            throw new Exception("Stack is full.");
        }

        // if Q1 is empty enqueue item to Q1
        if (Q1.isEmpty()) {
            Q1.enqueue(item);
        } else {
            // enqueue all elements to Q2
            while (!Q1.isEmpty()) {
                Q2.enqueue(Q1.dequeue());             // O(n)
            }
            // enqueue item to Q1
            Q1.enqueue(item);
            // enqueue all elements from Q2 back to Q1
            while (!Q2.isEmpty()) {
                Q1.enqueue(Q2.dequeue());             // O(n)
            }
            // item is now correctly at top of the stack
        }
    }
```

enqueue() uses linked list implementation of queue to add an element to the end of the list, which is constant time (sets pointers head/tail to a new node).

dequeue() is also constant time by using linked list implementation, again simply reassigning head/tail nodes.

With these in mind, push() complexity is **O(n)** because these methods are being called for every element of the linked list.

Stack's pop operation using enqueue and dequeue:

```
public T pop() throws Exception {
        // First detects if the stack is empty
        if (Q1.isEmpty()) {
            throw new Exception("Stack is empty.");
        }
        return Q1.dequeue();                          // O(1)
    }
```

Stack pop() only needs dequeue() because this queue operation aligns with stacks, removing and returning the topmost element.

As discussed, dequeue() takes constant time, and because pop() only calls this method once its complexity is also constant, **O(1)**.