Level: Medium

Implement a Queue using 2 stacks.

Questions to Clarify:
Q. Can we use extra memory?
A. Use only constant memory - O(1)

Q. What to return if the user tries to dequeue an empty queue?
A. Throw an exception.

Solution:
Queue is *First In First Out (FIFO)* whereas Stack is *Last In First Out (LIFO)*.

Let's say that we inserted item *A* on Stack *1*. Then, we inserted a bunch of other items on the same stack.

Stack *1*: **`A -> B -> C -> D    <--- top of stack`**
Stack *2*: **`Empty`**

To function like a queue, we need to return item *A* before other items.
Item *A* will be at the bottom. We need to bring it back to the top. We can do this using the second stack. If we flush all items one by one from stack *1* onto stack *2*, item *A* will now be on top.

After flushing
Stack *1*: **`Empty`**
Stack *2*: **`D -> C -> B -> A    <--- top of stack`**

So, if we want to return the element inserted first, we need to flush all elements into Stack *2*, then return the top of stack *2*.

We don't need to flush more elements into Stack *2* until it is empty.

Here is the algorithm for that:

```
enqueue(a):
   push a to s1

dequeue():
   if s2 is empty:
     flush from s1 to s2
   if s2 is still empty
     queue is empty, return nothing

   return s2.pop()
```

The time complexity for enqueue is O(1) since all we do is insert into *s1*.

The time complexity for a single dequeue is O(n), because in the worst case, we might flush *s1* into *s2*.

However, the overall time for *N* dequeues is *N* because we flush a total of *N* items.
So each dequeue operation is O(1) amortized time.

If we want O(1) worst case time in dequeues, we can perform the flush in the enqueue() function instead of the dequeue function. This flips the time complexity of enqueue and dequeue.

You can discuss these tradeoffs with the interviewer.

Pseudocode:

```
(Note: Never write pseudocode in an actual interview. Unless you're writing a few
lines quickly to plan out your solution. Your actual solution should be in a real
language and use good syntax.)

enqueue(a):
   push a to s1

dequeue():
   if s2 is empty:
     flush from s1 to s2
   if s2 is still empty
     queue is empty, return nothing

   return s2.pop()
```

Test Cases:
Edge Cases: Enqueue with empty, Dequeue with empty
Base Cases: Enqueue with 1 element, dequeue with 1 element
Regular Cases: Dequeue with stack 2 empty, Multiple dequeues/enqueues in a row

Time Complexity:

Enqueue: O(1)
Dequeue: O(n) in the worst case, O(1) amortized

Space Complexity: O(1) extra space
This is because we allocate 2 new stack pointers, but we move around nodes from the queue. So the only new memory we're allocating are the 2 pointers.

Now, this will depend on the Stack library you're using. Some libraries will not move the nodes around and create a new node every time you push on to the stack. But make sure you mention to the interviewer that we can do this in O(1) space by moving nodes around.

```
public class Queue<A> {
    Stack<A> s1;
    Stack<A> s2;

    public Queue() {
        s1 = new Stack<A>();
        s2 = new Stack<A>();
    }

    public void enqueue(A a) {
        s1.push(a);
    }

    public A dequeue() throws EmptyQueueException {
        if (s2.isEmpty()) {
            flushToS2();
        }
        if (s2.isEmpty())
            throw new EmptyQueueException();

        return s2.pop();
    }

    private void flushToS2() {
        while (!s1.isEmpty()) {
            s2.push(s1.pop());
        }
    }
}

/*
 * We created a new exception, which is really easy to do.
 * It also makes the code more readable above. We know exactly what the
 * error is.
 */
public class EmptyQueueException extends Exception {
    public EmptyQueueException() {
    }
}
```

Level: Medium

Use an array to implement 2 stacks.

Questions to Clarify:
Q. Is the array of fixed size?
A. Yes, the size is provided as input.

Q. What to return if the array is full?
A. Throw an exception.

Q. We will implement push() and pop() functions of the stack, is there any other function you want?
A. No, those two are enough.

Solution:
Each stack should support a push() and pop() operation.

We can simply use the 2 ends of the array as each stack. We keep adding elements at the 2 ends until they meet. At that point, we've run out of space.

Pseudocode:

```
(Note: Never write pseudocode in an actual interview. Unless you're writing a few
lines quickly to plan out your solution. Your actual solution should be in a real
language and use good syntax.)


a = new array[size]
// stack pointers represent where the next item on that stack will go
int stack1 = 0, stack2 = a.length - 1;
push(stack, item):
   if (stack1 > stack2 or stack1 >= a.length or stack2 < 0)
       array is full, throw exception
   a[stack] = item
   update stack value

pop(stack)
   if(stack is stack1 && stack1 == 0 or stack is stack2 && stack2 == a.length -
1)
         stack is empty, throw exception
   result = a[stack]
   update stack value
   return result
```

Test Cases:
Edge Cases: Array empty, array full, *s1* empty/full, *s2* empty/full
Base Cases: Single element in *s1*/*s2*, array size 0,1,2
Regular Cases: Array size 10

Time Complexity: O(1) for both push() and pop()

Space Complexity: O(1) - no extra space used apart from the array.

However, if the interviewer wants to count the array in the space complexity, it will be O(n).

```
public class ArrayStack {
    int[] a;

    int s1;
    int s2;

    public ArrayStack(int arraySize) {
        a = new int[arraySize];
        s1 = 0;
        s2 = a.length - 1;
    }

    public void push(int stackNumber, int item) throws StackFullException {
        if (stackNumber != 1 && stackNumber != 2)
            throw new IllegalArgumentException("Invalid stack number.");

        if (s1 > s2)
            throw new StackFullException();

        if (stackNumber == 1)
            a[s1++] = item;
        else
            a[s2--] = item;
    }

    public int pop(int stackNumber) throws StackEmptyException {
        if (stackNumber != 1 && stackNumber != 2)
            throw new IllegalArgumentException("Invalid stack number.");

        if (stackNumber == 1 && s1 > 0)
            return a[--s1];
        else if (stackNumber == 2 && s2 < a.length - 1)
            return a[++s2];

        throw new StackEmptyException();
    }
}

public class StackFullException extends Exception {
    public StackFullException() {
    }
}

public class StackEmptyException extends Exception {
    public StackEmptyException() {
    }
}
```