1 Illustrate the result of *each* of the operations `PUSH(S, 4)`, `PUSH(S, 1)`, `PUSH(S, 3)`, `POP(S)`, `PUSH(S, 8)`, and `POP(S)` on an initially empty stack *S* stored in array *S[0..5]*. Indicate where the `STACK_TOP` index is pointing.

Solution:

	Index						
	0	1	2	3	4	5	
PUSH(*S*, 4)	4						
STACK_TOP	↑						
PUSH(*S*, 1)	4	1					
STACK_TOP		↑					
PUSH(*S*, 3)	4	1	3				
STACK_TOP			↑				
POP(*S*)	4	1	3				returns 3
STACK_TOP		↑					
PUSH(*S*, 8)	4	1	8				
STACK_TOP			↑				
POP(*S*)	4	1	8				returns 8
STACK_TOP		↑					

2 Illustrate the result of each of the operations `ENQUEUE(Q, 4)`, `ENQUEUE(Q, 1)`, `ENQUEUE(Q, 3)`, `DEQUEUE(Q)`, `ENQUEUE(Q, 8)`, and `DEQUEUE(Q)` on an initially empty queue Q stored in array $Q[0..5]$. Here, `ENQUEUE` refers to inserting an element into the queue, and `DEQUEUE` refers to removing an element from the queue. Indicate where the `FRONT` and `REAR` pointers are.

Solution: Several solutions are possible. Here I assume that `REAR` is always pointing to the location where `ENQUEUE` would add an item and `FRONT` is always pointing to the location where `DEQUEUE` would remove an item. Therefore, initially `FRONT = REAR = 0`, which indicates an empty queue. Other implementations are possible.

	Index						
	0	1	2	3	4	5	
ENQUEUE(Q, 4)	4						
FRONT	↑						
REAR		↑					
ENQUEUE(Q, 1)	4	1					
FRONT	↑						
REAR			↑				
ENQUEUE(Q, 3)	4	1	3				
FRONT	↑						
REAR				↑			
DEQUEUE(Q)	4	1	3				returns 4
FRONT		↑					
REAR				↑			
ENQUEUE(Q, 8)	4	1	3	8			
FRONT		↑					
REAR					↑		
DEQUEUE(Q)	4	1	3	8			returns 1
FRONT			↑				
REAR					↑		

3 Show how to implement a queue using two stacks with the following primitives of the stack abstract data type: `EMPTY(S)`, `STACK TOP(S)`, `PUSH(S, element)`, and `POP(S)`? Assuming `O(1)` time complexity for all these primitives, what is the run-time complexity of each of the following queue operations: `EMPTY(Q)`, `ENQUEUE(Q, element)`, `DEQUEUE(Q)`, `FRONT(Q)`, `REAR(Q)`. (Note: Not all queue operations can be performed in `O(1)` time complexity.)

Solution: Here is one of several solutions to the problem. Let `S1` and `S2` be the two stacks. Use `S1` for storing the data in the queue and `S2` for temporary storage.

```
EMPTY(Q) = EMPTY(S1)
ENQUEUE(Q, element) = PUSH(S1, element)
REAR(Q) = STACK_TOP(S1)
DEQUEUE(Q):
   temp ← POP(S1)
   while not EMPTY(S1) {
      PUSH(S2, temp)
      temp ← POP(S1)
   }
   while not EMPTY(S2) {
      PUSH(S1, POP(S2))
   }
   return temp
FRONT(Q):
   while not EMPTY(S1) {
      PUSH(S2, POP(S1))
   }
   temp ← STACK_TOP(S2)
   while not EMPTY(S2) {
      PUSH(S1, POP(S2))
   }
   return temp
```

Note that the routines `DEQUEUE(Q)` and `FRONT(Q)` did not check for the precondition that the queue is non-empty. The assumption is that the primitives `POP(S1)` and `STACK TOP(S2)` would check for that precondition.

Clearly, `EMPTY(Q)`, `ENQUEUE(Q, element)`, and `REAR(Q)` are `O(1)`. `DEQUEUE(Q)` and `FRONT(Q)` are `O(n)`, where *n* is the number of elements in the queue.