

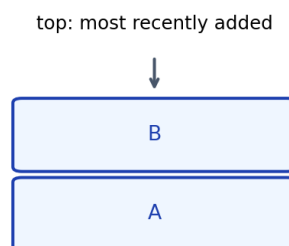
Solutions

4.1 Stack Behavior

Reference example

```
Deque<String> stack = new ArrayDeque<String>();  
stack.push("A"); stack.push("B");  
System.out.println(stack.pop());  
System.out.println(stack.peek());
```

Stack before the first pop



The next pop removes B because the most recent item is on top.

4.1.1 - Explain

Starting empty, push 2, push 5, pop, push 7. What values are removed by two more pops?

The first pop removes 5. After pushing 7, the next pops remove 7 and then 2.

4.1.2 - Write code

Reverse the values 1, 2, 3 using an ArrayDeque as a stack.

Push each value, then pop until empty.

```
Deque<Integer> s = new ArrayDeque<Integer>();  
for (int v : new int[]{1,2,3}) s.push(v);  
while (!s.isEmpty()) System.out.print(s.pop() + " ");
```

Expected output

3 2 1

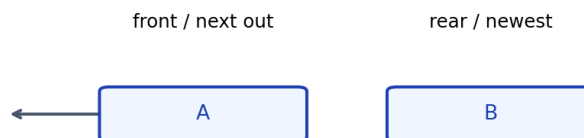
Solutions

4.2 Queue Behavior

Reference example

```
Queue<String> queue = new ArrayDeque<String>();
queue.offer("A"); queue.offer("B");
System.out.println(queue.poll());
System.out.println(queue.peek());
```

Queue before the first poll



The next poll removes A because it entered first.

4.2.1 - Explain

Starting empty, enqueue 4, enqueue 9, dequeue, enqueue 2. Give the remaining dequeue order.

The first dequeue removes 4. The remaining order is 9, then 2.

4.2.2 - Write code

Use a queue to print 3, 1, 4 in their insertion order.

Append at the rear and remove from the front.

```
Queue<Integer> q = new ArrayDeque<Integer>();
q.offer(3); q.offer(1); q.offer(4);
while (!q.isEmpty()) System.out.print(q.poll() + " ");
```

Expected output

3 1 4

Solutions

4.3 Implementation Choices

Reference example

Array-backed stack: end insertion/removal
Linked queue with head and tail: endpoint updates
Array front removal with shifting: linear work
ArrayDeque: circular-array deque

4.3.1 - Explain

Why is repeatedly removing index 0 from an ArrayList a poor queue implementation for large inputs?

Each removal shifts later elements left, taking linear time in the current list size. A suitable deque avoids that repeated shifting.

4.3.2 - Explain

What additional reference makes appending to a singly linked queue constant-time?

A maintained tail reference identifies the last node without scanning from the head. Empty-list insertion must update both head and tail.

Solutions

4.4 Amortised Reasoning

Reference example

Dynamic array capacities: 1, 2, 4, 8, ...
A full array triggers allocation and copying
Most appends write one element
An individual resize append may cost $O(n)$

4.4.1 - Explain

When growing capacity by doubling, how many elements are copied across expansions from capacities 1, 2, and 4?

The copies total $1 + 2 + 4 = 7$. The geometric sum remains proportional to the number of appended elements.

4.4.2 - Explain

Why is constant amortised time different from constant worst-case time?

Amortised time bounds the average cost across a sequence. It permits an occasional expensive operation, so it does not promise every individual operation is constant-time.