Solutions

Each answer shows the steps, then the **result**.

2.1

- a **stack** is **LIFO**: items are added and removed at the same end, the top, so the last item added is the first removed
- a **queue** is **FIFO**: items are added at the rear and removed from the front, so the first item added is the first removed

2.2

- `Rear <- (Rear MOD 6) + 1`
- when `Rear` is 6: $(6 \bmod 6) + 1 = \mathbf{1}$, so the pointer wraps back to cell 1

2.3

- the free list is a chain of the array's **unused** slots, headed by its own pointer
- a slot is taken from it when a node is inserted and returned to it when a node is deleted, so the program always knows where there is room

2.4

- before adding: check the queue is **not full** – that there is an unused element
- before removing: check the queue is **not empty** – that there is at least one item

2.5

(a) a **stack**

(b) a **queue**

3.1

(a) `Rear <- (6 MOD 6) + 1 = 1`, so `Rear` becomes **1**

- the item is stored in **cell 1**: the pointer has wrapped round to the start

(b) `Front` moves on twice: $(4 \bmod 6) + 1 = 5$, then $(5 \bmod 6) + 1 = 6$, so `Front` is **6**

- `NumberInQueue` was 4 after the addition, so after two removals it is **2**

(c) the queue is full when `NumberInQueue` equals `MaxSize`, that is 6

3.2

- in a linear queue both pointers only ever move **forward**, so once `Front` has passed a cell that cell can never be reused
- the queue therefore reports itself full when `Rear` reaches the end of the array, even though the cells before `Front` are empty
- a circular queue wraps a pointer back to cell 1 when it passes the last cell, using `(Pointer MOD MaxSize) + 1`, so every cell is reused and the queue is full only when it really holds `MaxSize` items

3.3

- take the slot at `FreeListHead`, index **3**, and move the head on: `FreeListHead <- Pointer[3]`, which is **4**
- store the value in `Data[3]`
- `Pointer[3] <- Pointer[2]`, which is **5**, so the new node points at what the previous node pointed at
- `Pointer[2] <- 3`, so the previous node now points at the new one
- the list is $2 \to 3 \to 5 \to 1$ and the free list is $4 \to 6$
- the order matters: setting `Pointer[2]` before `Pointer[3]` destroys the address of the rest of the list

3.4

- find the node **before** index 5, which is index 3, by following the pointers from `Start`
- `Pointer[3] <- Pointer[5]`, which is **1**, so the list bypasses the deleted node
- return the slot to the free list: `Pointer[5] <- FreeListHead`, which is 4
- `FreeListHead <- 5`, so the free list is now $5 \to 4 \to 6$ and the list is $2 \to 3 \to 1$
- the data in `Data[5]` is not erased; the node is deleted because nothing points at it any more

3.5

- a **linked list**
- patients are seen in arrival order, and a linked list keeps them in that order by following the pointers from the head
- a patient may leave before being seen, and deleting a node from the middle of a linked list changes only pointers, with no shifting of the other items, which an array would need; and the list can grow as far as memory allows, which matters because the number waiting is not known in advance
- a plain queue is accepted **only** if the answer deals with removal from the middle; a queue removes from the front only, so a patient leaving early cannot be taken out of it