

Exercise walk-through

10.4.1

**The circular queue, a linked list
in arrays, and choosing the right
ADT**

A-Level Computer Science

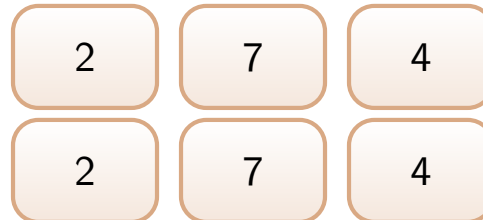
Arrays

An **array** holds many values of the **same** type under one name, reached by an **index**. A **record** is the opposite: different types, always belonging together.

1-D

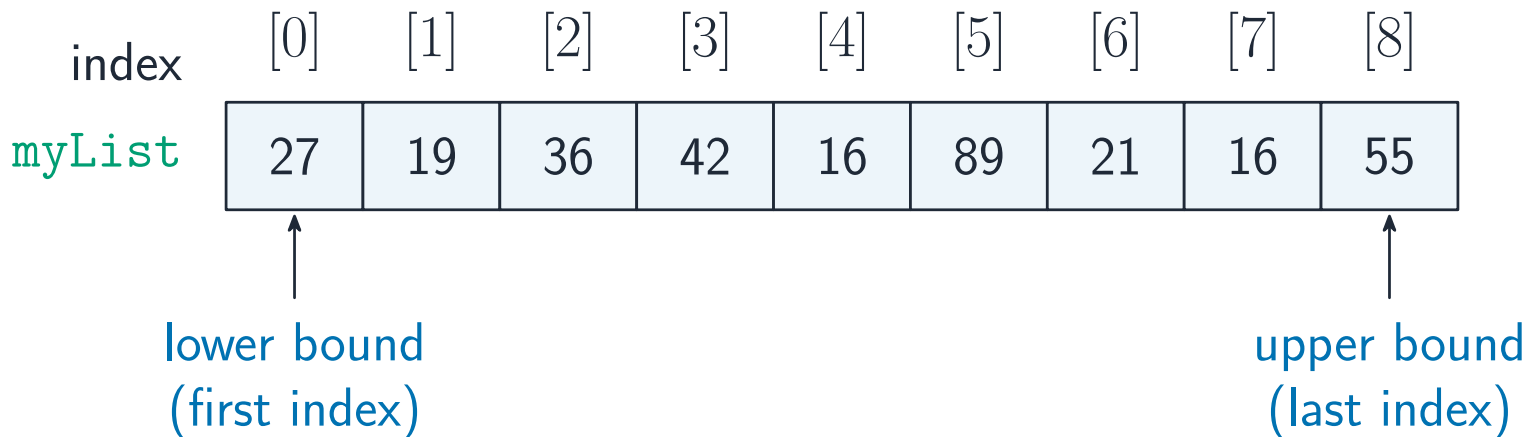


2-D – [row, column]



1-D arrays

One index. Marks [3] is the third element – but check the **lower bound**: Cambridge pseudocode usually starts at 1, most languages at 0.



Array vocabulary, and the standard sweep

index

the position of an element within the array

bounds

the **lowest and highest valid indices** – going outside them is an error

dimension

how many indices are needed: 1-D for a single sequence, 2-D for a table

a 2-D array

the **first** index is the row, the **second** the column; **nested loops** visit every cell

to traverse

visit every element in turn

the sweep pattern

to find a sum, count, maximum or minimum, set a **running variable** before the loop and update it inside

The running variable must be initialised **outside** the loop. Inside, it resets every pass – the single commonest bug in this topic.

2-D arrays: a table

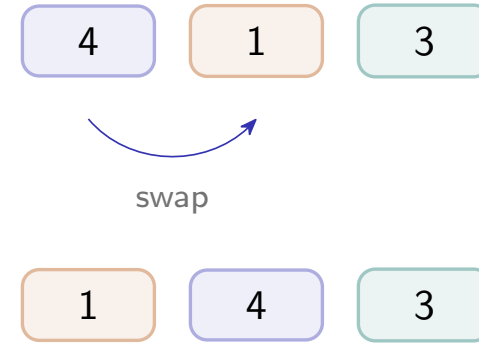
Two indices, **row first**: `Grid[2,5]` is row 2, column 5.

		column index			
		$[r, 1]$	$[r, 2]$	$[r, 3]$	$[r, 4]$
row index	$[1, c]$	12	31	17	48
	$[2, c]$	19	67	99	29
	$[3, c]$	42	22	95	61

`Grid[2,3]`
(row 2, column 3)

Searching and sorting

Linear search walks from the start until it finds the target – works on any list, $O(n)$. **Bubble sort** swaps adjacent pairs that are out of order, so the largest “bubbles” to the end each pass – $O(n^2)$.



Exercise 2.1

2.1 State **one** key feature of a **stack** and **one** key feature of a **queue**.

[2]



stack



queue

Exercise 2.1

2.1 State **one** key feature of a **stack** and **one** key feature of a **queue**.

[2]



stack



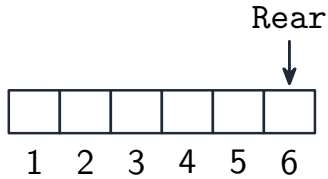
queue

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

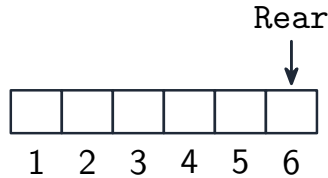
Exercise 2.2

2.2 A circular queue is held in an array with `MaxSize = 6`. Give the statement that moves `Rear` on by one, and state the value it gives when `Rear` is 6. [2]



Exercise 2.2

2.2 A circular queue is held in an array with `MaxSize = 6`. Give the statement that moves `Rear` on by one, and state the value it gives when `Rear` is 6. [2]



2.2

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

Exercise 2.3

2.3 State what is meant by the **free list** in a linked list held in arrays.

[2]

	Data	Pointer
1		
2		
3		
4		
5		
6		

Exercise 2.3

2.3 State what is meant by the **free list** in a linked list held in arrays.

[2]

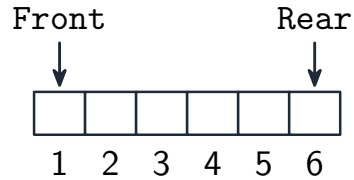
	Data	Pointer
1		
2		
3		
4		
5		
6		

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

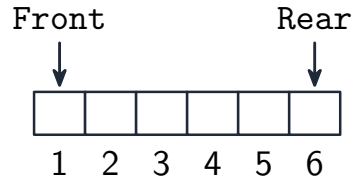
Exercise 2.4

2.4 State the check that must be made **before** adding an item to a queue, and the check that must be made **before** removing one. [2]



Exercise 2.4

2.4 State the check that must be made **before** adding an item to a queue, and the check that must be made **before** removing one. [2]



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

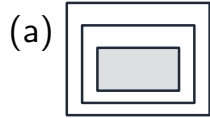
Exercise 2.5

2.5 Name the most suitable ADT for each situation.

[2]

(a) storing the return addresses of nested procedure calls

(b) holding documents waiting to be printed



(b)



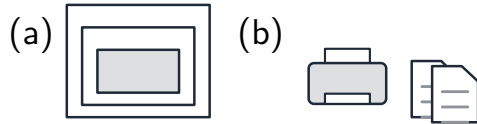
Exercise 2.5

2.5 Name the most suitable ADT for each situation.

[2]

(a) storing the return addresses of nested procedure calls

(b) holding documents waiting to be printed



2.5

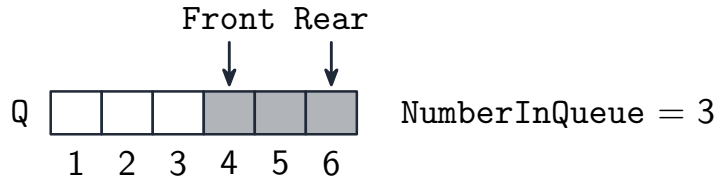
(a) a **stack**

(b) a **queue**

Exercise 3.1 (a)

3.1 A circular queue is held in $Q[1:6]$, with $\text{Front} = 4$, $\text{Rear} = 6$ and $\text{NumberInQueue} = 3$.

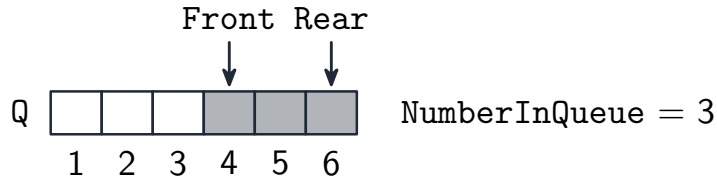
(a) An item is added. Give the new value of **Rear**, and state which cell the item is stored in. [2]



Exercise 3.1 (a)

3.1 A circular queue is held in $Q[1:6]$, with $\text{Front} = 4$, $\text{Rear} = 6$ and $\text{NumberInQueue} = 3$.

(a) An item is added. Give the new value of **Rear**, and state which cell the item is stored in. [2]



(a) $\text{Rear} \leftarrow (6 \text{ MOD } 6) + 1 = 1$, so **Rear** becomes **1**

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

Exercise 3.1 (b)

3.1 A circular queue is held in $Q[1:6]$, with $\text{Front} = 4$, $\text{Rear} = 6$ and $\text{NumberInQueue} = 3$.

(b) Two items are then removed. Give the new value of Front and of NumberInQueue .^[2]

Exercise 3.1 (b)

3.1 A circular queue is held in $Q[1:6]$, with **Front** = 4, **Rear** = 6 and **NumberInQueue** = 3.

(b) Two items are then removed. Give the new value of **Front** and of **NumberInQueue**.^[2]

(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**

Exercise 3.1 (c)

3.1 A circular queue is held in $Q[1:6]$, with $\text{Front} = 4$, $\text{Rear} = 6$ and $\text{NumberInQueue} = 3$.

(c) State how the program knows the queue is **full**, given that it keeps NumberInQueue .^[1]

Exercise 3.1 (c)

3.1 A circular queue is held in $Q[1:6]$, with $\text{Front} = 4$, $\text{Rear} = 6$ and $\text{NumberInQueue} = 3$.

(c) State how the program knows the queue is **full**, given that it keeps NumberInQueue .^[1]

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

Exercise 3.2

3.2 Explain why a **linear** queue held in an array wastes space, and how a **circular** queue avoids it. [3]



1 2 3 4 5 6

Exercise 3.2

3.2 Explain why a **linear** queue held in an array wastes space, and how a **circular** queue avoids it. [3]



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 $(\text{Pointer} \bmod \text{MaxSize}) + 1$, so every cell is reused and the queue is full only when it really holds **MaxSize** items

Exercise 3.3

3.3 A linked list is held in `Data[1:6]` and `Pointer[1:6]`. `Start = 2`, the list is $2 \rightarrow 5 \rightarrow 1$, `Pointer[1] = 0`, and the free list is $3 \rightarrow 4 \rightarrow 6$ with `FreeListHead = 3`.

Describe the steps that insert a new value between the nodes at index 2 and index 5, giving the new value of every pointer that changes. [4]

	Data	Pointer	
1		0	
2		5	Start = 2
3		4	
4		6	FreeListHead = 3
5		1	
6		0	



Exercise 3.3

3.3 A linked list is held in `Data[1:6]` and `Pointer[1:6]`. `Start = 2`, the list is $2 \rightarrow 5 \rightarrow 1$, `Pointer[1] = 0`, and the free list is $3 \rightarrow 4 \rightarrow 6$ with `FreeListHead = 3`.

Describe the steps that insert a new value between the nodes at index 2 and index 5, giving the new value of every pointer that changes. [4]

	Data	Pointer	
1		0	
2		5	Start = 2
3		4	
4		6	FreeListHead = 3
5		1	
6		0	

3.3

- take the slot at `FreeListHead`, index **3**, and move the head on: `FreeListHead <-`

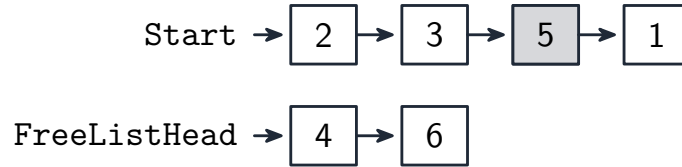
Full solution 3.3

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 $2 \rightarrow 3 \rightarrow 5 \rightarrow 1$ and the free list is $4 \rightarrow 6$
- the order matters: setting `Pointer[2]` before `Pointer[3]` destroys the address of the rest of the list

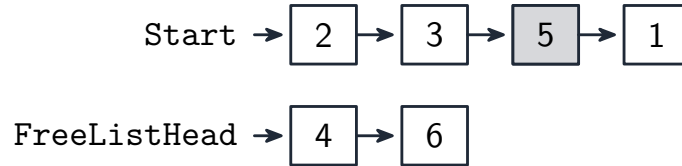
Exercise 3.4

3.4 Using the list as it is **after** the insertion in 3.3, describe the steps that **delete** the node at index 5, including what happens to its slot. [4]



Exercise 3.4

3.4 Using the list as it is **after** the insertion in 3.3, describe the steps that **delete** the node at index 5, including what happens to its slot. [4]



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 $5 \rightarrow 4 \rightarrow 6$ and the list is $2 \rightarrow 3 \rightarrow 1$
- the data in `Data[5]` is not erased; the node is deleted because nothing points at it any more

Exercise 3.5

3.5 A hospital records patients waiting for a doctor. Patients are seen in the order they arrive, but a patient may leave before being seen, and the number waiting is not known in advance.

Name a suitable ADT and **justify** your choice.

[3]



Exercise 3.5

3.5 A hospital records patients waiting for a doctor. Patients are seen in the order they arrive, but a patient may leave before being seen, and the number waiting is not known in advance.

Name a suitable ADT and **justify** your choice.

[3]



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