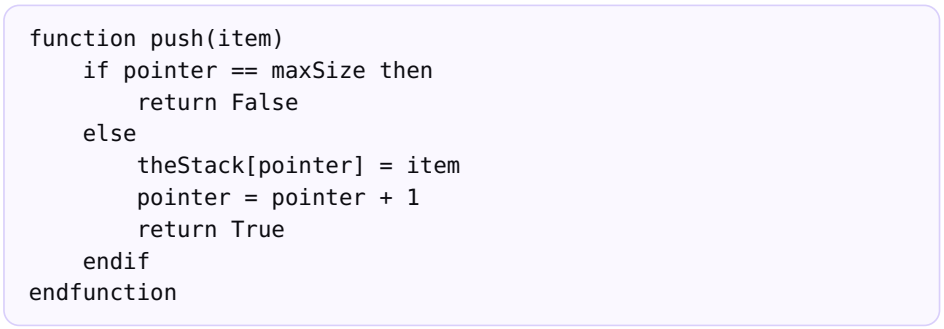
1.4 DATA TYPES & STRUCTURES · 1.4.2

Stacks & queues — Mark scheme

56 marks · spec 1.4.2(b)(c)

AO key: AO1 = knowledge · AO2 = application · AO3 = programming. Code answers are indicative — award for correct logic in any consistent language. Traces verified.

Q	ANSWER	AO	MARKS
1(a)	9. (1)	AO2	1
1(b)	<table><tr><td>INDEX</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>VALUE</td><td>6</td><td>1</td><td>4</td><td>–</td><td>–</td></tr></table> index 0 = 6 and index 1 = 1 (1); index 2 = 4 (overwrites the popped 9) (1); pointer = 3 (1). (3)	AO2	3
1(c)	Application (1) + justification (1), e.g. an undo feature — the most recent action must be reversed first (LIFO); or the call stack / browser back button / RPN evaluation / backtracking. (2)	AO1	2

Q	ANSWER	AO	MARKS
2	 Full test `pointer == maxSize` + `return False` (1); store at `theStack[pointer]` (1); increment pointer after storing (1); `return True` in else branch + valid structure (1). (4)	AO3	4

Q	ANSWER	AO	MARKS

3(a)	`function pop()`
	`  if topStack == 0 then`
	`  return "EMPTY"`
	`  else`
	`  topStack = topStack - 1`
	`  item = numbers[topStack]`
	`  return item`
	`  endif`
	`endfunction`
	
	Empty test + return `"EMPTY"` (1); decrement `topStack` then read `numbers[topStack]` (1); return the item + valid structure (1). (3)
		AO3	3

3(b)	`topStack` points at the next free space, not the top item, so it must be decremented to point at the actual top item before reading. (1)	AO1	1

Q	ANSWER	AO	MARKS

4	`for i = 0 to 7`
	`  value = input("Enter a character")`
	`  push(value)`
	`next i`
	`for i = 0 to 7`
	`  data = pop()`
	`  print(data)`
	`next i`
	
	First loop iterates 8 times (1); input inside the loop (1); `push(value)` called (1); second loop of 8 calling `pop()` (1); output the popped value (1). (5)
		AO3	5

Q	ANSWER	AO	MARKS
5*	1 mark per point, max 4: on a call, the return address (next instruction in the caller) is pushed onto the call stack; parameters / local variables / register contents may also be pushed (a stack frame); execution jumps to the subroutine's code; on return the address is popped and execution resumes at the correct point; LIFO order lets nested / recursive calls return in the correct (reverse) order. (4)	AO1	4

Q	ANSWER	AO	MARKS
6(a)	"A" removed → `headPointer = 3` (1); `tailPointer = 5` (unchanged) (1). (2)	AO2	2
6(b)	"D" stored at index 5 → `tailPointer = 6` (1); it is a linear queue, so when the tail reaches the end of the array it is treated as full (1); the freed spaces at indexes 0–1 cannot be reused because the pointers only move forward / there is no wrap-around (1). (3)	AO2	3

Q	ANSWER	AO	MARKS

7	`function enqueue(item)`
	`    if tailPointer == maxSize then`
	`        return False`
	`    else`
	`        theQueue[tailPointer] = item`
	`        tailPointer = tailPointer + 1`
	`        return True`
	`    endif`
	`endfunction`
	
	Full test + return `False` (1); store at `theQueue[tailPointer]` (1); increment `tailPointer` (1); return `True` + structure (1). (4)
		AO3	4

Q	ANSWER	AO	MARKS

8	`function dequeue()`
	`    if headPointer == tailPointer then`
	`        return "EMPTY"`
	`    else`
	`        item = theQueue[headPointer]`
	`        headPointer = headPointer + 1`
	`        return item`
	`    endif`
	`endfunction`
	
	Empty test `headPointer == tailPointer` (1); return `"EMPTY"` (1); read the front item (1); increment `headPointer` AND return the item (1). (4)
		AO3	4

Q	ANSWER	AO	MARKS
9	Line + correction (1 + 1, each), 3 errors: • Line 03: should be `return False` (wrong value returned when full). (2) • Line 05: should be `theQueue[tailPointer] = item` (wrong pointer used). (2) • Line 06: should be `tailPointer = tailPointer + 1` (must increment, not decrement). (2)	AO2	6

Q	ANSWER	AO	MARKS
10	Linear: pointers only move forward; reaching the end of the array = full, even if front spaces are free (1); circular: pointers wrap around to index 0 (modulo arithmetic) so freed spaces are reused (1); advantage: more efficient use of memory / no wasted spaces / no need to shuffle elements down (1). (3)	AO1	3

Q	ANSWER	AO	MARKS
11	A standard queue is FIFO — items are removed strictly in the order they were added (1); a high-priority job added later would have to wait behind earlier jobs, so it could not be processed first (1); a priority queue would be more suitable (items leave by priority, not arrival order) (1). (3)	AO2	3

Q	ANSWER	AO	MARKS

12*	```
	choice = input("Add, remove or quit?")
	while choice != "quit"
	    if choice == "add" then
	        jobName = input("Enter job name")
	        success = enqueue(jobName)
	        if success == False then
	            print("Queue is full")
	        else
	            print("Job added")
	        endif
	    elseif choice == "remove" then
	        result = dequeue()
	        if result == "EMPTY" then
	            print("Queue is empty")
	        else
	            print(result)
	        endif
	    endif
	    choice = input("Add, remove or quit?")
	endwhile
	``
	Loop continues until "quit" (1); input choice controls/re-prompts the loop (1); selection on "add" (1); input job name & call `enqueue` with it (1); check return value → "Queue is full" / "Job added" (1); selection on "remove" calling `dequeue()` (1); test for "EMPTY" → "Queue is empty" (1); otherwise output the removed job + working structure (1). (8)
		AO3	8

Total for worksheet: 56 marks