ANSWERS

A3.3 Queues: linear, circular and priority

Data structures · A level · OCR H446 1.4.2, AQA 7517 4.2.1.4, Eduqas A500QS 1.1 · about 20 min

What this lesson is about

Front and rear pointers, wrapping round with MOD, and priority queues: a command queue for the robot.

- The queue
- A linear queue
- The circular queue
- Priority queues
- Where queues are used

Questions

6 marks in all

1. What is the main advantage of a circular queue over a linear queue built on an array? [1 mark]

- ☐ **A** Slots freed at the front can be reused without moving any items
- ☐ **B** Items can be removed from either end
- ☐ **C** It never becomes full
- ☐ **D** Items leave in order of priority

> **Answer:** A. The pointers wrap round with MOD, so space freed by dequeues is used again, and no items are shuffled.

2. A circular queue has 5 slots, indexed 0 to 4, and rear is 4. After one more item is enqueued, what is rear? [1 mark]

> **Answer:** 0. rear = (4 + 1) MOD 5 = 0: the pointer wraps round to the start.

3. This program moves a circular queue's pointers for enqueue (E) and dequeue (D). What does it print? [1 mark]

```
SIZE = 4
front, rear, count = 0, -1, 0
for op in ["E", "E", "E", "D", "D", "E", "E"]:
    if op == "E":
        rear = (rear + 1) % SIZE
        count = count + 1
    else:
        front = (front + 1) % SIZE
        count = count - 1
print(front, rear, count)
```

> **Answer:**
>
> ```
> 2 0 3
> ```
>
> Three enqueues take rear to 2, two dequeues take front to 2, and the last two enqueues take rear to 3 and then wrap it to 0, with 3 items left.

4. In a priority queue, two items have the same priority. In which order do they leave? [1 mark]

- ☐ **A** In the order they arrived
- ☐ **B** In reverse order of arrival
- ☐ **C** In alphabetical order
- ☐ **D** In a random order

> **Answer:** A. Within one priority a priority queue behaves as an ordinary queue: first in, first out.

5. Which of these would normally use a queue? [1 mark]

Tick every answer that is true.

- ☐ **A** A keyboard buffer
- ☐ **B** Print jobs waiting for a printer
- ☐ **C** Undo in a text editor
- ☐ **D** Breadth-first search of a graph
- ☐ **E** Return addresses of subroutine calls

> **Answer:** A, B, D. Buffers, print queues and breadth-first search need first in, first out. Undo and return addresses need a stack.

6. Why does a circular queue built on an array usually keep a count of its items as well as front and rear? [1 mark]

- ☐ **A** Front and rear can be in the same relative positions for a full queue and an empty one
- ☐ **B** MOD cannot be used without a count
- ☐ **C** The count stores the item at the front
- ☐ **D** Without a count the queue could only hold one item

> **Answer:** A. When the queue fills, rear ends up just behind front, which is exactly where it is when the queue is empty. The count tells the two apart.

The task: the command queue

Build a static circular queue at the top level of the program: an array `queue` of `SIZE` elements (here **3**), with `front` starting at 0, `rear` at -1 and `count` at 0. Wrap both pointers with `%`. Do not use the list's own `append`, `pop` or `insert`. - Write `enqueue(item)`: if the queue is full return -1; otherwise store the item and return the index of the slot it went into. - Write `dequeue()`: if the queue is empty return `None`; otherwise remove the item at the front and return it. Each function changes pointers, so give it a `global` line naming the ones it changes. Then go through `EVENTS` in order: - `"add <command>"`: enqueue the command (the text after `add `). Print `added <command> at <slot>`, for example `added forward 20 at 0`, or `full, dropped <command>` if it did not fit. - `"run"`: dequeue a command, print `ran <command>`, and carry it out with `do(command)`. A dropped command must never be carried out.

```
# the two lines every program starts with: the commands, then the robot
from bugbot import *
connect()

def do(command):
    """command is a string such as "forward 20": a direction and a whole number of cm."""
    move, cm = command.split()
    cm = int(cm)
    if move == "forward":
        forward(50, distance=cm)
    elif move == "backward":
        backward(50, distance=cm)
    elif move == "left":
        left(50, distance=cm)
    elif move == "right":
        right(50, distance=cm)

SIZE = 3
queue = [None] * SIZE
front = 0
rear = -1
count = 0

def enqueue(item):
    return -1

def dequeue():
    return None

EVENTS = ["add forward 20", "add right 20", "run", "add forward 15", "add left 10",
          "add backward 50", "run", "run", "run"]
```

The hint students can ask for: Work out what happens to rear when it is already at the last slot: it must come back to 0. A count of items is the easiest way to tell a full queue from an empty one, because front and rear alone can look the same for both.

A solution

```
# the two lines every program starts with: the commands, then the robot
from bugbot import *
connect()

def do(command):
    move, cm = command.split()
```

```
    cm = int(cm)
    if move == "forward":
        forward(50, distance=cm)
    elif move == "backward":
        backward(50, distance=cm)
    elif move == "left":
        left(50, distance=cm)
    elif move == "right":
        right(50, distance=cm)

SIZE = 3
queue = [None] * SIZE
front = 0
rear = -1
count = 0

def enqueue(item):
    global rear, count
    if count == SIZE:
        return -1
    rear = (rear + 1) % SIZE
    queue[rear] = item
    count = count + 1
    return rear

def dequeue():
    global front, count
    if count == 0:
        return None
    item = queue[front]
    front = (front + 1) % SIZE
    count = count - 1
    return item

EVENTS = ["add forward 20", "add right 20", "run", "add forward 15", "add left 10",
          "add backward 50", "run", "run", "run"]

for event in EVENTS:
    if event == "run":
        command = dequeue()
        print("ran", command)
        do(command)
    else:
        command = event[4:]
        slot = enqueue(command)
        if slot == -1:
            print("full, dropped", command)
        else:
            print("added", command, "at", slot)
```

Any program that meets the task's checks is marked correct in the simulator; this is one way, not the only way.