TEMASEK JUNIOR COLLEGE

**Temasek Junior College**
**2022 JC1 H2 Computing**
**Data Structures 3 – Queues**

## §3.1 Introduction to Queues

A **queue** is an abstract data type that holds an ordered, linear sequence of data elements. It is a **first-in-first-out (FIFO)** data structure, where the first data element added to the data structure is the first data element to be removed.

For 9569 H2 Computing, we shall be taking a look at:
- Linear queues
- Circular queues

## §3.2 Queue Pointers

In a queue, new elements are added to the back or rear of the queue as the last element. When an element is removed, the remaining elements **do not** move up to take the empty space.

To implement a queue and maintain its order, you need to maintain a **front pointer** (head) that always points to the front of the queue. When the queue is empty, the **front pointer** typically pointers to the first empty slot of the queue. When the queue contains data elements, the **front pointer** will always point to the first data element.

In addition, there needs to be a **rear pointer** (tail) that points to the last element of the queue.

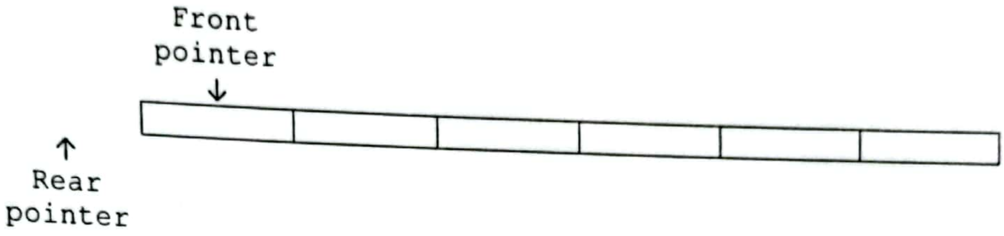
## §3.3 Queue Operations

The main queue operations are given in the table below.

| Keyword | Operation |
|---|---|
| `ENQUEUE(<DATA>)` | Adds a data element to the rear of the queue. |
| `DEQUEUE()` | Returns the element currently at the front of the queue. |
| `IS_EMPTY()` | Checks whether a queue is empty. |
| `IS_FULL()` | Checks whether a queue is at maximum capacity when implemented as a static (fixed-size) structure.<br><br>In a **linear queue**, it checks whether the **rear pointer** is pointing to the end of the queue, regardless whether every slot of the queue is occupied. You may use `IS_END()` instead to make explicit that the function is meant to check whether the **rear pointer** is pointing to the end of the queue. |
| `DISPLAY()` | Outputs the current elements including their order in the queue. |

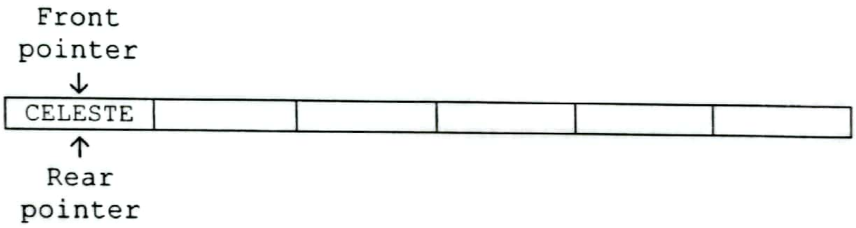
**Example 1**

- This is a simplified example of a queue in use.
- The queue in this example can only store six data elements.
- The front pointer is used to show where the current first element of the queue is. In an empty queue, it points to the first slot.
- The rear pointer is used to show where the current last element of the queue is.
- The queue is currently empty.



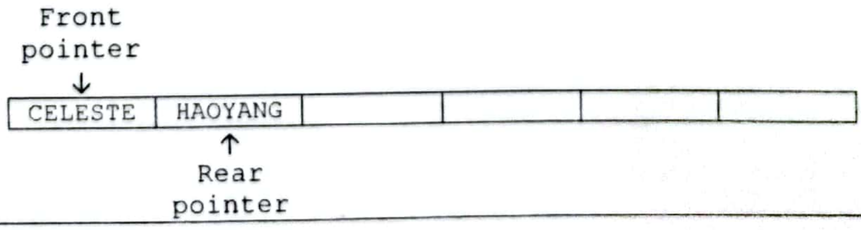

- The first data element CELESTE is added to the queue.
- The front pointer and the rear pointer is now pointing at the same data element. This happens when there is only one data element in the queue.

```
ENQUEUE(CELESTE)
```




- Continue to add four more data elements, HAOYANG, JUN JIE, LE QI, and WEIZHI, one after another to the queue.
- The front pointer continues pointing at CELESTE, the first data element in the queue.
- The rear pointer moves along the queue to point at the element at the rear of the queue.

```
ENQUEUE(HAOYANG)
```



`ENQUEUE(JUN JIE)`

Front pointer ↓

| | | | | | |
|---|---|---|---|---|---|
| CELESTE | HAOYANG | JUN JIE | | | |

↑ Rear pointer

`ENQUEUE(LE QI)`

Front pointer ↓

| | | | | | |
|---|---|---|---|---|---|
| CELESTE | HAOYANG | JUN JIE | LE QI | | |

↑ Rear pointer

`ENQUEUE(WEIZHI)`

Front pointer ↓

| | | | | | |
|---|---|---|---|---|---|
| CELESTE | HAOYANG | JUN JIE | LE QI | WEIZHI | |

↑ Rear pointer

- Remove `CELESTE` from the front of the queue.
- The front pointer moves to point to `HAOYANG`. This will show that `CELESTE` is no longer in the queue.
- The data does not **need** to be deleted as it will no longer be part of the queue.

`DEQUEUE()`

Front pointer ↓

| | | | | | |
|---|---|---|---|---|---|
| ~~CELESTE~~ | HAOYANG | JUN JIE | LE QI | WEIZHI | |

↑ Rear pointer

## §3.4 Linear Queues

What you have seen in Example 1 is a **linear queue**, where you can envisage the data as a line. The first item in is the first item out. The maximum size of the queue in **Example 1** is fixed i.e. static in this case, although it could be dynamic.

A typical method for storing data in a queue is to use a one-dimensional array. In this section, we shall look at the static implementation of a linear queue using an array.

The `frontPointer` is initialised to `1` and the `rearPointer` to `0`. The `rearPointer` will be incremented every time you add an item to the queue, so this initial value will ensure that the first item is added into the first position of the array, position `1`. Once the first item has been added, both `frontPointer` and `rearPointer` will correctly point to the first element in the array.

Free space at the front is not reused in a linear queue. Elements will not be added when the `rearPointer` is at the end of the array, no matter how many items there are in the queue.

| Process | Pseudocode | |
|---|---|---|
| Setting up a queue | `DECLARE queue ARRAY[1 : n] OF INTEGER`<br>`DECLARE global rearPointer : INTEGER`<br>`DECLARE global frontPointer : INTEGER`<br>`DECLARE global queueEnd : INTEGER`<br><br>`frontPointer ← 1`<br>`rearPointer ← 0`<br>`queueEnd ← LENGTH(ARRAY)` | |
| Defining the `IS_END()` function | `FUNCTION IS_END()`<br>`    IF rearPointer = queueEnd`<br>`        THEN`<br>`            RETURN TRUE`<br>`        ELSE`<br>`            RETURN FALSE`<br>`    ENDIF`<br>`ENDFUNCTION` | When the `rearPointer` is of the same value as `queueEnd`, it is pointing to the end of the queue. |
| Defining the `IS_EMPTY()` function | `FUNCTION IS_EMPTY()`<br>`    IF rearPointer < frontPointer`<br>`        THEN`<br>`            RETURN TRUE`<br>`        ELSE`<br>`            RETURN FALSE`<br>`    ENDIF`<br>`ENDFUNCTION` | When the `rearPointer` is lesser than the `frontPointer`, the queue is empty. |
| Defining `DISPLAY()` function | `FUNCTION DISPLAY()`<br>`    IF IS_EMPTY() = TRUE`<br>`        THEN`<br>`            OUTPUT("Queue is empty")`<br>`        ELSE`<br>`            OUTPUT(queue[frontPointer :`<br>`                   rearPointer])` | |

| | | |
|---|---|---|
| Adding data element to the queue | <pre>PROCEDURE ENQUEUE(queue, data)<br>    IF IS_END() = TRUE<br>        THEN<br>            OUTPUT("Not added, end of queue")<br>        ELSE<br>            rearPointer = rearPointer + 1<br>            queue[rearPointer] ← data<br>    ENDIF<br>ENDPROCEDURE</pre> | Before adding an item to the queue you need to ensure that the queue is not full.<br><br>When `rearPointer` is pointing to the final slot of the array, there is no room to add new items.<br><br>`IS_FULL()` can be defined to carry out this check.<br><br>If the end of the array has not been reached, the `rearPointer` is incremented and the new element is added to the queue. |
| Removing data element from front of queue | <pre>FUNCTION DEQUEUE(queue)<br>    IF IS_EMPTY() = TRUE<br>        THEN<br>            OUTPUT("Queue is empty")<br>            RETURN NULL<br>        ELSE<br>            dq_Item ← queue[frontPointer]<br>            frontPointer ← frontPointer + 1<br>            RETURN dq_Item<br>    ENDIF<br>ENDFUNCTION</pre> | Before taking an element from the queue you need to check that the queue is not empty.<br><br>`IS_EMPTY()` can be used to check whether the `rearPointer` is less than the `frontPointer`.<br><br>If the queue is not empty, the element at the front of the queue (as referenced by `frontPointer`) is returned and `frontPointer` is incremented by 1.<br><br>`queue[frontPointer]` does not need to be deleted as it will no longer be part of the queue. |

**Question**

In a stack, the space freed up from popping a data element on the top of the stack can be overwritten with a new data element.

What happens to the space freed up from the removal of a data element at the front of the queue? Since data is added at the rear of the queue, can you then re-use the space at the front of the queue?

If no, why? If yes, how?

**Exercise 1**

Write Python code to implement a linear queue as a class in your Jupyter notebook.

## §3.5 Circular Queues

A static array implementation of a queue has a fixed capacity. As you add elements to the queue, you will eventually reach the end of the array.

When items are dequeued, space is freed up at the start of the array. To move remaining items in the queue forward to occupy the freed-up space can be time consuming and cumbersome. Every element will have to move if this was done. A more efficient way to do so is to implement a **circular queue** (also called a **circular buffer**) .

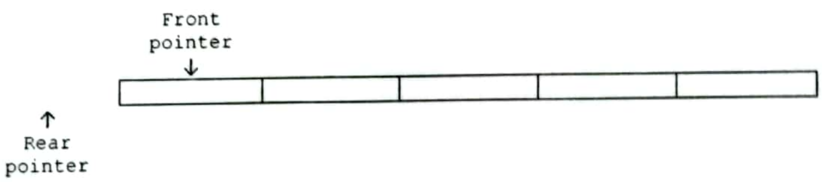
A **circular queue** is a queue that reuses the empty slots at the front of the array that arises when elements are removed from the queue. It is a FIFO data structure implemented as a ring where the front and rear pointers can wrap around from the end to the start of the array. Think of a circular queue as a fixed-size ring where the back of the queue is connected to the front.

As items are continuously enqueued, the rear pointer will eventually reach the last position of the array. It then **wraps around** to point to the start of the array (so long as the array is not full).

Similarly, as items are dequeued, the front index pointer will wrap around until it passes the rear index pointer (which shows that the queue is empty). As with a linear queue, it is the pointers that move rather than the data.

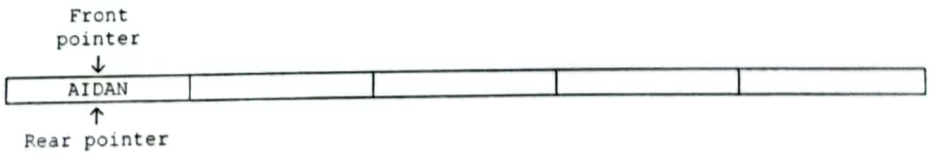
**Example 2**

- This is a simplified example of a circular queue in use.
- The queue in this example can only store five data elements.
- The front pointer is used to show where the current first element of the queue is. In an empty queue, it points to the first slot.
- The rear pointer is used to show where the current last element of the queue is.
- The queue is currently empty.

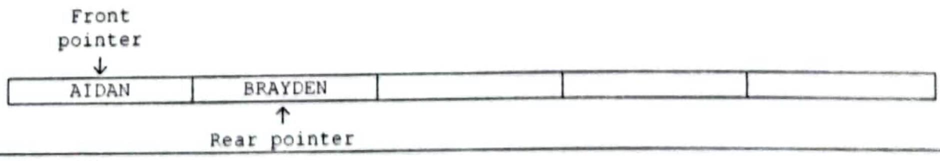



- Add four data elements one by one to the queue.

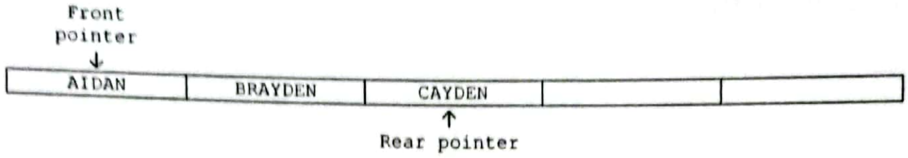
`ENQUEUE(AIDAN)`



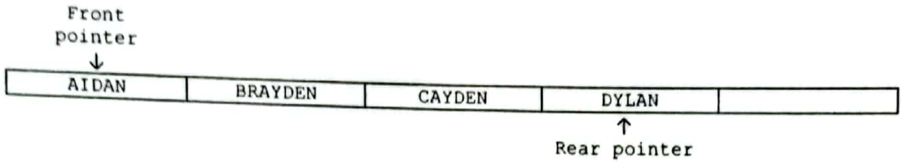

`ENQUEUE(BRAYDEN)`



ENQUEUE(CAYDEN)

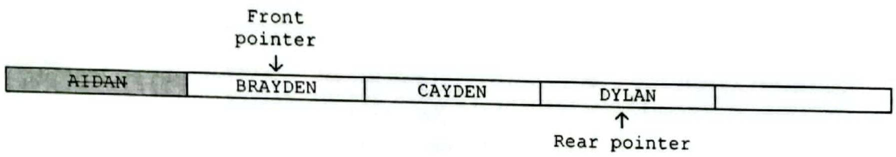



ENQUEUE(DYLAN)

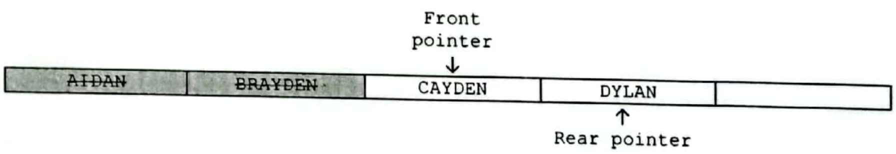



- Remove the first two data elements in the queue.

DEQUEUE()

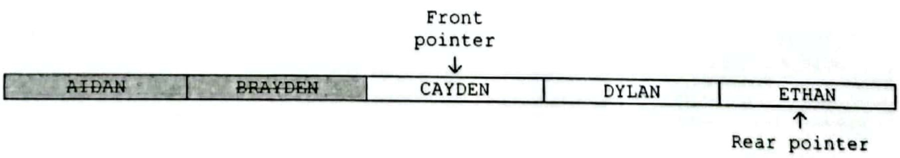



DEQUEUE()

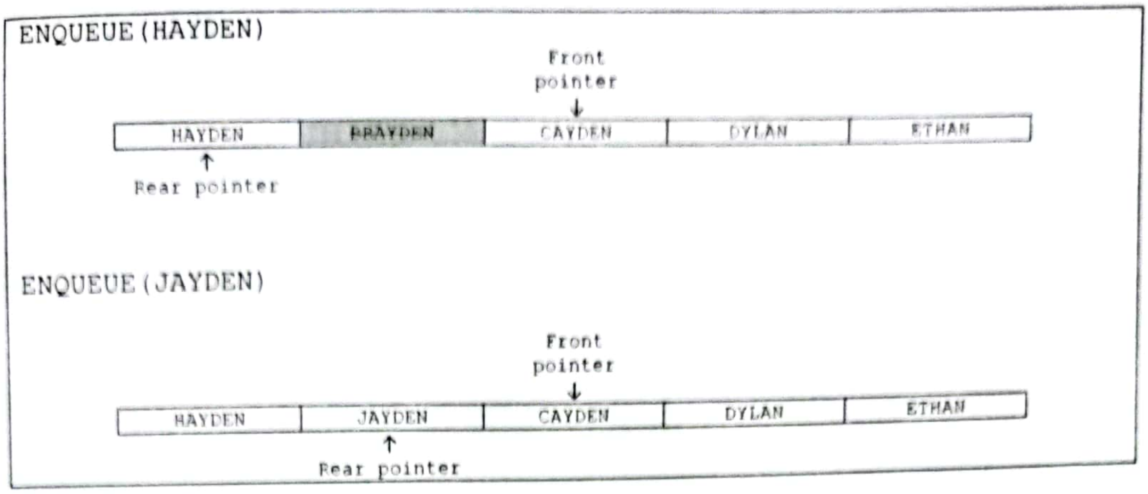



- Add one data element to the queue.
- This element will be placed at the end of the queue.

ENQUEUE(ETHAN)




- Add one more data element to the queue.
- Since the rear pointer is already at the end of the queue, it **wraps around** to point to the start of the array.
- The existing copy of data is overwritten




## §3.6 Static Array Implementation of Circular Queues

As with linear queues, a **front pointer** `frontPointer` and a **rear pointer** `rearPointer` needs to be maintained when implementing a circular queue.

The `frontPointer` is initialised to `1` and the `rearPointer` to `0`. The `rearPointer` will be incremented every time you add an item to the queue, so this initial value will ensure that the first item is added into the first position of the array, position `1`. Once the first item has been added, both `frontPointer` and `rearPointer` will correctly point to this element in the array.

In a circular queue, free space arising from removal of data elements from the front of the queue can be reused. Hence there is a need to be able to advance the index pointers in such a way that they will reset to the start of the array once the end of the array has been reached.

| Process | Pseudocode | |
|---|---|---|
| Setting up a queue | `DECLARE queue ARRAY[1 : n] OF INTEGER`<br>`DECLARE global rearPointer : INTEGER`<br>`DECLARE global frontPointer : INTEGER`<br>`DECLARE global queueEnd : INTEGER`<br><br>`frontPointer ← 1`<br>`rearPointer ← 0`<br>`queueEnd ← LENGTH(ARRAY)`<br>`queueLength ← 0` | The setup is same as that of linear queue.<br><br>However, an additional variable `queueLength`, which represents the number of data elements currently in the queue is defined and initialised to `0`. |
| Defining the `IS_END()` function | `FUNCTION IS_END()`<br>`    IF rearPointer = queueEnd`<br>`        THEN`<br>`            RETURN TRUE`<br>`        ELSE`<br>`            RETURN FALSE`<br>`    ENDIF`<br>`ENDFUNCTION` | When the `rearPointer` is of the same value as `queueEnd`, it is pointing to the end of the queue. |
| Defining the `IS_FULL()` function | `FUNCTION IS_FULL()`<br>`    IF queueLength = queueEnd`<br>`        THEN`<br>`            RETURN TRUE`<br>`        ELSE`<br>`            RETURN FALSE`<br>`    ENDIF`<br>`ENDFUNCTION` | When the `queueLength` is of the same value as `queueEnd`, the queue has reached its maximum allowed capacity. |

| Process | Pseudocode | |
|---|---|---|
| Defining the IS_EMPTY() function | FUNCTION IS_EMPTY()<br>IF queueLength = 0<br>THEN<br>RETURN TRUE<br>ELSE<br>RETURN FALSE<br>ENDIF<br>ENDFUNCTION | When the queueLength is 0, there is nothing in the queue. |
| Defining DISPLAY() function | FUNCTION DISPLAY()<br>IF IS_EMPTY() = TRUE<br>THEN<br>OUTPUT("Queue is empty")<br>ELSE<br>OUTPUT(queue[frontPointer : rearPointer]) | Note that you have to implement an equivalent code to output queue[frontPointer:rearPointer] in the programming language of your choice. |
| Adding data element to the queue | PROCEDURE ENQUEUE(queue, data)<br>IF IS_FULL() = TRUE<br>THEN<br>OUTPUT("QUEUE IS FULL")<br>ELSE<br>IF IS_END() = TRUE<br>THEN<br>rearPointer ← 1<br>ELSE<br>rearPointer ← rearPointer + 1<br>ENDIF<br>queue[rearPointer] ← data<br>queueLength ← queueLength + 1<br>ENDIF<br>ENDPROCEDURE | Before adding an item to the queue you need to ensure that the queue is not full.<br><br>IS_FULL() can be defined to carry out this check.<br><br>When rearPointer is pointing to the final slot of the array, it needs to be reset to the start. The element is then added.<br><br>If the end of the array has not been reached, the rearPointer is incremented and the new element is added to the queue. |
| Removing data element from front of queue | FUNCTION DEQUEUE(queue)<br>IF IS_EMPTY() = TRUE<br>THEN<br>OUTPUT("Queue is empty")<br>RETURN NULL<br>ELSE<br>dq_Item ← queue[frontPointer]<br>queueLength ← queueLength - 1<br>IF frontPointer = queueEnd<br>THEN<br>frontPointer ← 1<br>ELSE<br>frontPointer ← frontPointer + 1<br>ENDIF<br>RETURN dq_Item<br>ENDIF<br>ENDFUNCTION | Before taking an element from the queue you need to check that the queue is not empty.<br><br>IS_EMPTY() can be used to check whether the rearPointer is less than the frontPointer.<br><br>If the queue is not empty, the element at the front of the queue (as referenced by frontPointer) is returned and frontPointer is incremented by 1.<br><br>queue[frontPointer] does not need to be deleted as it will no longer be part of the queue. |

## Exercise 2

Write Python code to implement a circular queue as a class in your Jupyter notebook.

## §3.7 Circular Queues vs. Linear Queues

Use of a circular queue brings with it advantages over a linear queue.

- ***Easier for insertion-deletion***
  In the circular queue, elements can be inserted easily if there are vacant locations until it is fully occupied, whereas in the case of a linear queue insertion is not possible once the rear pointer reaches the end of the queue even if there are empty locations present in the queue.

- ***Efficient utilisation of memory***
  In the circular queue, there is no wastage of memory as it uses the unoccupied space. Hence allocated memory can be effectively used compared to a linear queue.

- ***Ease of performing operations***
  In the linear queue, FIFO is followed, so the element inserted first is the element to be deleted first. This is not the scenario in the case of the circular queue as the rear and front are not fixed so the order of insertion-deletion can be changed, which is very useful.

## §3.8 Priority Queues (Extra - Not in Syllabus)

A **priority queue** is a variation of a FIFO structure where some data may leave out of sequence when it has a higher priority than other data items. A **priority queue** adds a further element to the queue which is the priority of each item. Items in a **priority queue** are then removed according to their order of priority. When two items are of the same priority, the item that is first added to the queue will be the item that is removed first. This adheres to the FIFO principle of queues.

Consider documents being sent to print on a network printer. It might be possible for the print manager program to control the queue in some way. They may be able to force print jobs to the top of the queue or to put print jobs on hold, whilst others are completed first.

This is known as a 'priority' queue and requires the programmer to assign priority ratings to different jobs. Higher priority jobs are effectively able to jump the queue. Where two jobs have the same priority, they will be handled according to their position in the queue i.e. the job that is added to the queue first will be completed first (FIFO).

## §3.9 Static Array Implementation of Priority Queues (Extra - Not in Syllabus)

A priority queue can also be implemented using an array by assigning a value to each element to indicate the priority. Items of data with the highest priority are dealt with first. Where the priority is the same, then the items are dealt with on a FIFO basis like a normal queue.

There are two possible ways to implement priority queues using a static array.

**Method 1**

- Use a standard queue where items are added in the usual way at the end of the queue.
- When items are removed, each element in the array is checked for its priority to identify the next item to be removed.
- Where this method is used, adding data is straightforward but removing it is more complex.

**Method 2**

- Maintain the queue in priority order, which means that when a new item is added, it is put into the correct position in the queue.
- Removing items can then be done in the usual way by taking the item at the front of the queue.
- Where this method is used, removing data is straightforward but adding it is more complex.

**References**

Isaac Computer Science - Queues
https://isaaccomputerscience.org/concepts/dsa_datastruct_queue?examBoard=all&stage=all&topic=data_structures