Queue

definitions are highlighted!

Introduction

- *A collection of sequential data in which new data is added to the end of the queue while data can only be removed from the front*
- A First In First Out (FIFO) data structure
    - The first entry placed in the queue will be the first to be removed from it, and vice versa.
- Each data stays in its storage location until its turn to be removed comes.
- There are 2 pointers — 'front' and 'rear' — indicating the front and end of the queue respectively.
- Logical view:

An example of a logical view of a queue with 5 items:

LOCATION	101	102	103	104	105	106	107	108	109	110
CONTENT	20	4	8	12	11					

Front Pointer ↑ Rear Pointer ↑

- Operations:
    - Enqueue (item): Add a new item to the end of the queue.
    - Dequeue (): Remove the front item from the queue and return it.
    - IsEmpty ()
    - IsFull ()
    - Size ()
- Overflow occurs when an attempt is made to add data to a queue when all available locations are occupied.
- Underflow occurs when an attempt is made to remove data from an empty queue.

Usage

- Buffer — used in the buffer area of main storage to handle data for input/output operations
    - This allows the processor to handle input and output operations efficiently without being delayed by the slower speeds of peripheral devices. Thus, the data is processed in a fair and ordered manner, maintaining the integrity of the input/output operations.
    - e.g. Keystrokes are stored in an input buffer while typing on a keyboard
    - e.g. Data is stored in an output buffer when sending it to output devices (e.g. printer)
- Spooling — output waiting to be printed is stored in a queue on a disk.
    - Used in networked environments where multiple computers share a single printer
    - The disk-based queue provides persistent storage, allowing print jobs to be retained even in the event of system shutdown and crashes.
- Queue-based simulations
    - Help to understand and optimise queuing time in real-life systems, improving operational efficiency in the real world

Linear queue

- Can be implemented using 3 variables: front, rear and size
    - When the queue is empty, rear is initialized to null.
- Insertion:

```
PROCEDURE Enqueue(item)
    IF rear = maxIndex THEN write ("Queue Overflow")
    ELSE
        rear = rear + 1
        queue[rear] = item
        size = size + 1
    ENDIF
ENDPROCEDURE
```

- Deletion:

```
FUNCTION Dequeue
    IF size = 0 THEN write ("Queue Underflow")
    ELSE
        deleted_item = queue[front]
        front = front + 1
        size = size - 1
        RETURN deleted_item
    ENDIF
ENDFUNCTION
```

- Memory is not reused:
    - *Example*: Enqueue(99)

Index	0	1	2	3	4	5	6	7	8	9
Content	20	4	8	12	11	88	32	56	99	

- **front** = 0 **rear** = 8 **size** = 9

Example: Dequeue

Index	0	1	2	3	4	5	6	7	8	9
Content	20	4	8	12	11	88	32	56	99	

- **front** = 1 **rear** = 8 **size** = 8

Example: Enqueue(77)

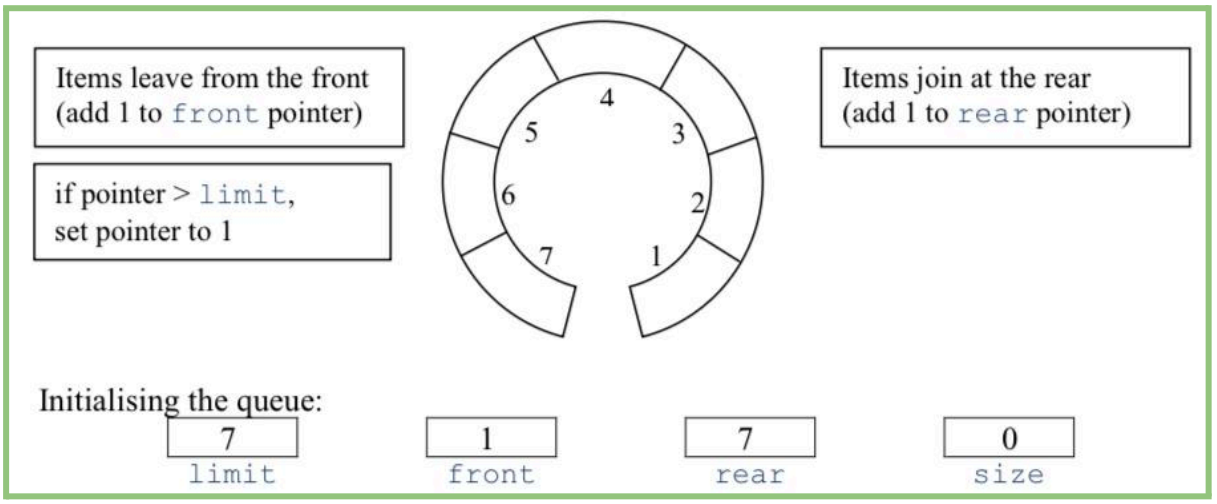
Index	0	1	2	3	4	5	6	7	8	9
Content	20	4	8	12	11	88	32	56	99	77

- **front** = 1 **rear** = 9 **size** = 9

    - At this point, all the locations in the queue are occupied; no additional items can be added although the first item has been removed.

- One way to overcome this problem is to shift each item to the left by 1 space each time an item is dequeued. However, this will slow down the processing time and worsen the time complexity.
- A better solution is to use a circular queue.

Circular queue

- A fixed area of storage for the queue is allocated and the rear of the queue “wraps around” to the start
- 

  - The first item is held in a location which is logically next to the storage location for the last item of the queue.
  - The rear of the queue is initialised to limit, because this element immediately precedes the first element.
    - The rear pointer is moved forward whenever a new element joins the queue. [see pseudocode for enqueuing]
    - This is different from a linear queue where the rear pointer is initialised to null.

```
PROCEDURE enqueue(item)
    IF size = limit THEN write ('Queue Overflow') //check for overflow
    ELSE
        IF rear = limit THEN rear = 1          //advance the rear pointer
        ELSE rear = rear + 1
        ENDIF
        queue[rear] = item                     //store item into rear
        size = size + 1                        //increment size
    ENDIF
ENDPROCEDURE
```

```
FUNCTION dequeue
    IF size = 0 THEN write ('Queue Underflow') //check for underflow
    ELSE
        deleted_item = queue[front]    //store front item into deleted_item
        size = size - 1                //decrement size
        IF front = limit THEN front = 1
        ELSE front = front + 1         //advance the front pointer
        ENDIF
        RETURN deleted_item
    ENDIF
ENDFUNCTION
```

Implementations

- Linear queue (array-based; static)
    - Fixed-size structure where elements are added to the rear and removed from front.
    - Memory is not reused when elements are removed from the front unless the entire array is shifted, which is inefficient.
- Circular queue (array-based)
    - Enhances the linear queue by reusing memory locations at the front of the array; when the rear reaches the end, it wraps around to the beginning
    - Slightly more complex pointer handling, but no shifting is needed
- Linked list queue (dynamic)
    - Each node contains the data and a pointer to the next node.
    - New nodes are added to the rear while nodes are removed from the front.
    - Both the front and rear pointers are updated to point to the appropriate nodes during enqueue and dequeue operations.

Feature	Linear queue	Circular queue	Linked list queue
Memory usage	Fixed size; unused spaces are not reused	Reuses memory efficiently by wrapping around	Dynamic; grows and shrinks as needed
Overflow conditions	When rear reaches the end of the array	When size = limit	Does not occur unless memory runs out
Reusability	No (unless elements are shifted)	Yes	Yes (nodes are created and deleted dynamically)
Pointer movement	Front and Rear pointers only move forward.	Front and Rear pointers wrap to the beginning.	Front and Rear move to newly created or deleted nodes dynamically.
Implementation complexity	Easiest to implement	Slightly harder to implement (due to the more complex logic)	Hardest to implement (due to the pointers)
Performance	Fast access using index, but slower is elements are shifted	More efficient as elements are not shifted	Pointer traversal is slow, but if the Front and Rear pointers are maintained, it is not needed.

@star.notes