COMP20230: Data Structures & Algorithms
Lecture 10: Queues

Dr Andrew Hines

Office: E3.13 Science East
School of Computer Science
University College Dublin

andrew.hines@ucd.ie

Data Structures

Operations	Array	List
`size, is_empty`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`get_elem_at_rank`	$\mathcal{O}(1)$	$\mathcal{O}(n)$
`set_elem_at_rank`	$\mathcal{O}(1)$	$\mathcal{O}(n)$
`insert_element_at_rank`	$\mathcal{O}(n)$	search $+ \mathcal{O}(1)$
`remove_element_at_rank`	$\mathcal{O}(n)$	search $+ \mathcal{O}(1)$
`insert_first, insert_last`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`insert_after, insert_before`	$\mathcal{O}(n)$	$\mathcal{O}(1)$

Implementations for different applications

Linked lists for flexible data

Arrays for fast access

Arrays vs Dynamic Arrays

Python Lists: dynamic arrays (not lists as the name would imply!)

Outline

- Queue (concept)
- Queue ADT
- Pseudo-code array based queue
- Pseudo-code Linked-list queue

Queue: concept

Queue: definition

- A queue's insertion and removal routines follow the first-in-first-out (FIFO) principle
- Elements may be inserted at any time, but only the element which has been in the queue the longest may be removed
- Elements are inserted at the rear (enqueued) and removed from the front (dequeued)

Real World Queue Examples

Direct applications

Waiting lists, bureaucracy

Access to shared resources (e.g., printer)

Indirect applications

Auxiliary (helper) data structure for algorithms

Component of other data structures (e.g. to enforce ordered processing)

Real World Queue Examples

How to ensure we process credit card payments in the correct order?

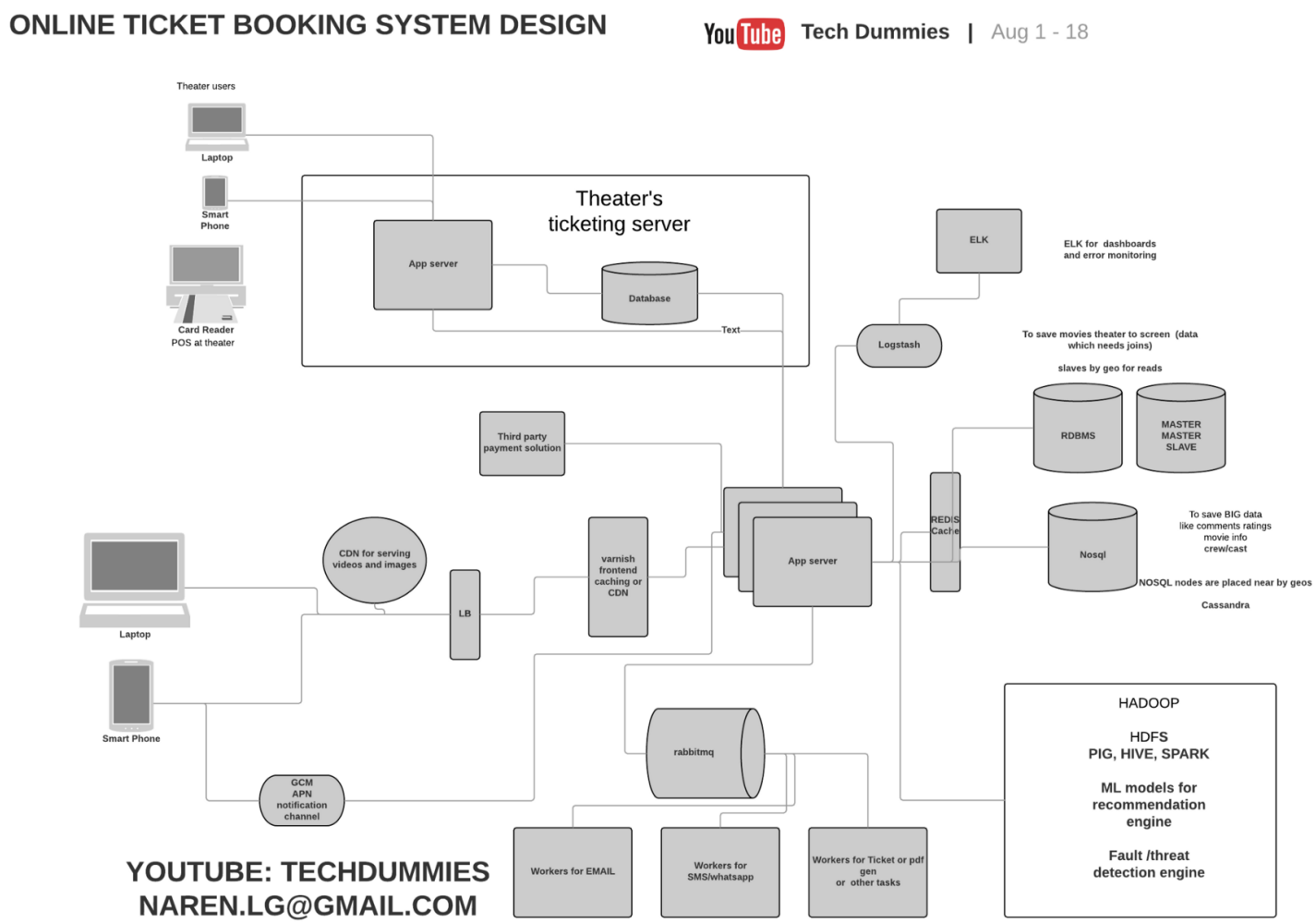


Image Credit: `https://medium.com/@narengowda/bookmyshow-system-design-e268fefb56f5`

Queue: ADT

The queue supports **two fundamental methods**:

`enqueue(o)`: Insert object o at the rear of the queue

Input: Object
Output: None

`dequeue()`: Remove the object from the front of the queue and return it; an error occurs if the queue is empty

Input: None
Output: Object

Queue: ADT

Support methods also need to be defined:

`size()`: Return the number of objects in the queue

Input: none
Output: integer

`is_empty()`: Return a boolean value that indicates whether the queue is empty.

Input: none
Output: boolean

`front()`: Return, but do not remove, the front object in the queue; an error occurs if the queue is empty.

Input: none
Output: Object

Queue: Implementing the ADT

Linked-list implementation

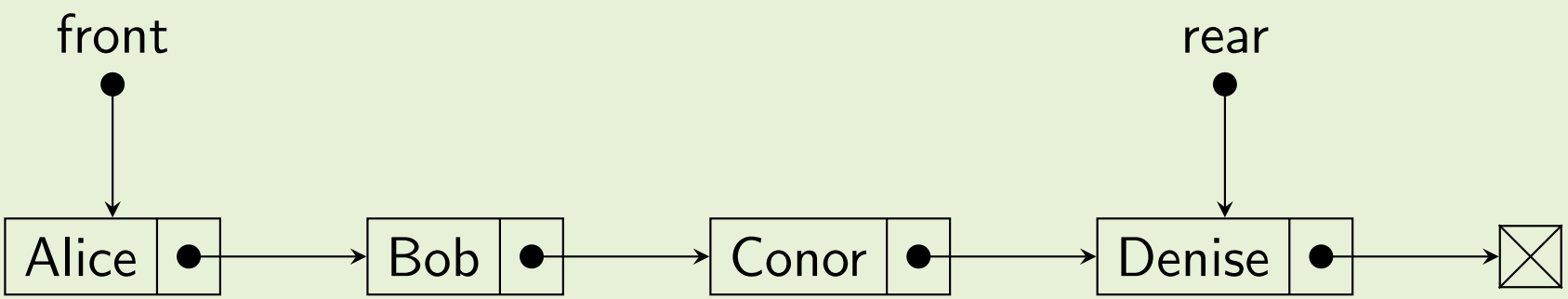
Seems the best option as there is no need to access elements in the middle of the queue
(That is what arrays are good at)

Queue: Implementing the ADT

Linked-list implementation

Seems the best option as there is no need to access elements in the middle of the queue
(That is what arrays are good at)

Queue needs 2 pointers



Alice is at the front of the queue (element 0) and Denise is at the rear (the last element).

Queue: Implementing the ADT

Linked-list implementation

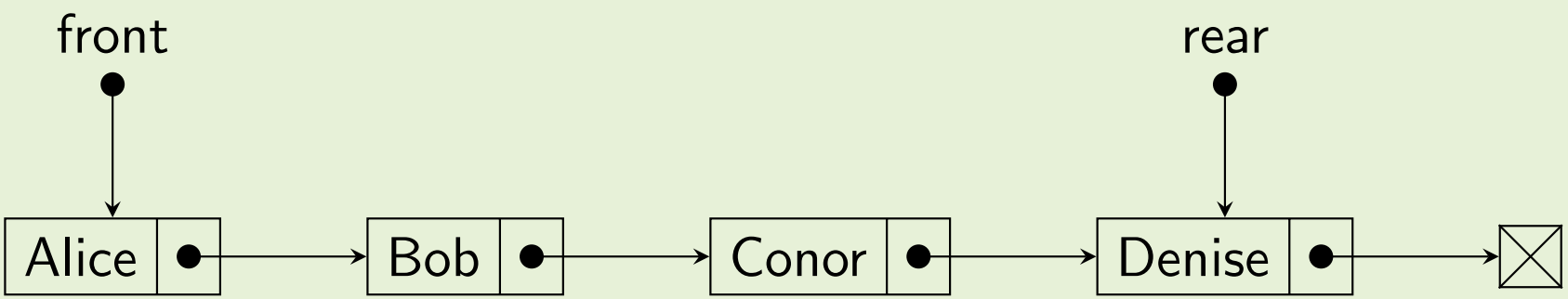
Seems the best option as there is no need to access elements in the middle of the queue
(That is what arrays are good at)

Queue: Implementing the ADT

Linked-list implementation

Seems the best option as there is no need to access elements in the middle of the queue
(That is what arrays are good at)

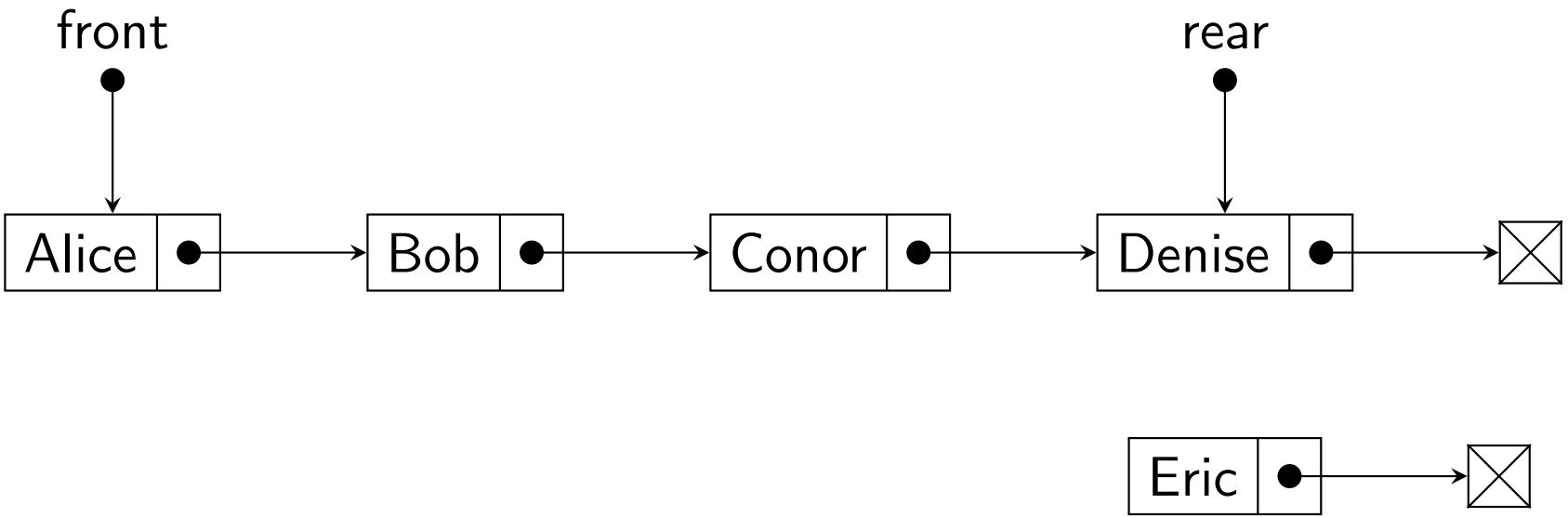
Queue: Enqueue



Alice is at the front of the queue (element 0) and Denise is at the rear (the last element).

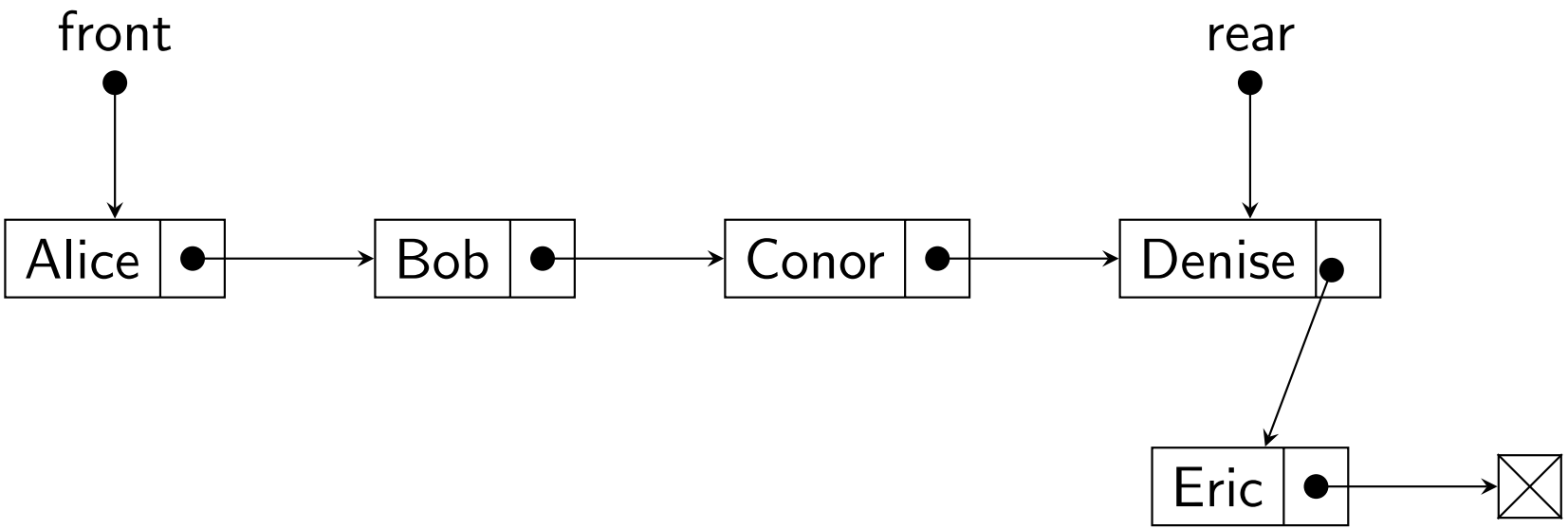
Enqueue

Queue has Alice, Bob, Conor and Denise in it.
Add Eric to the the queue
Eric joins the rear of the tail and the rear pointer moves to Eric



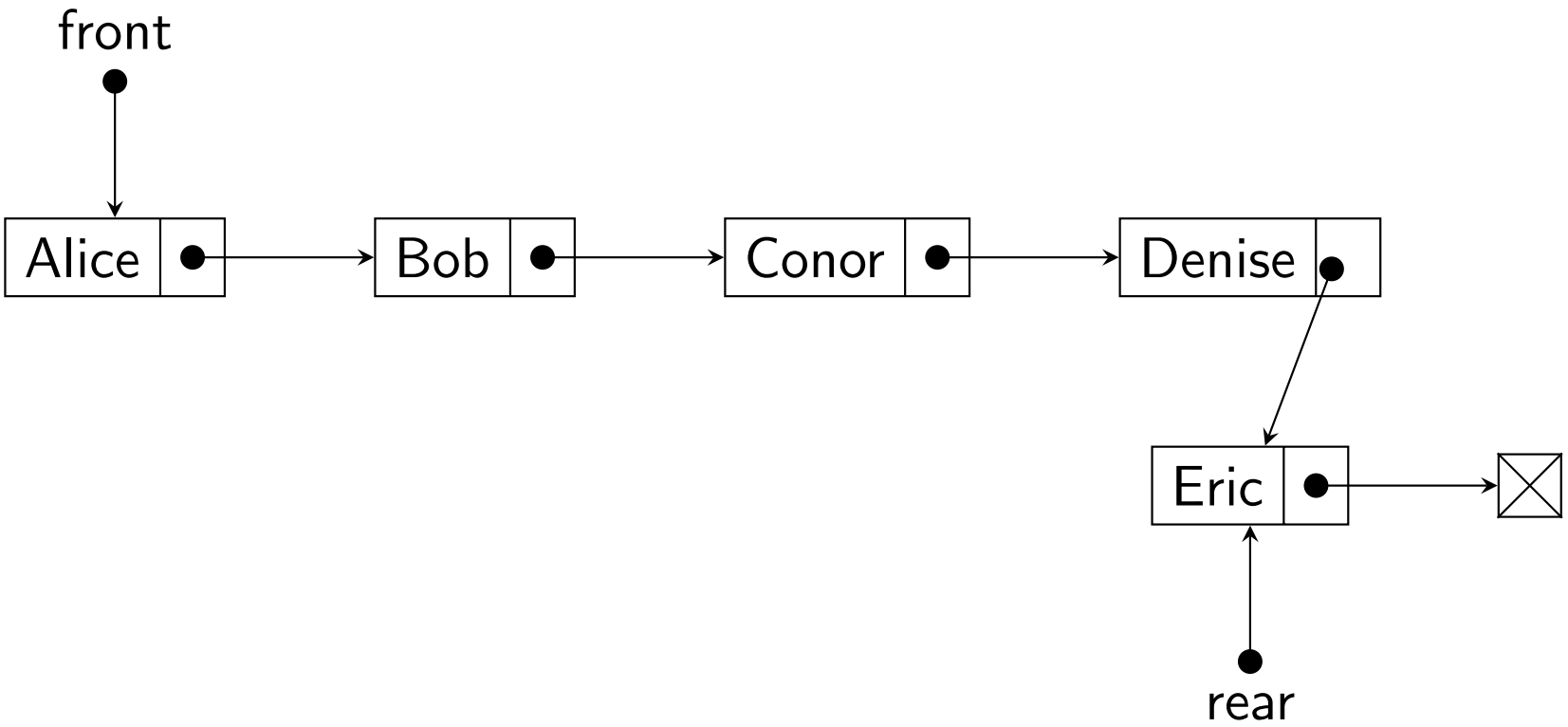
Enqueue

First we change the element.next() for the element at *rear* to point to Eric.



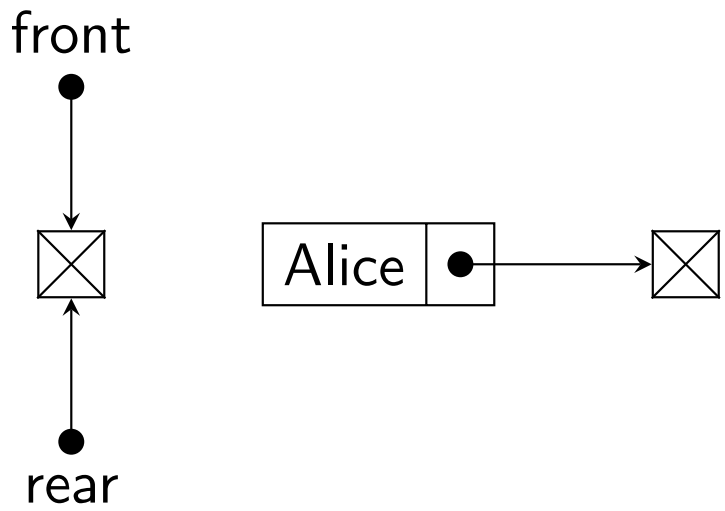
Enqueue

Then we change rear to point at Eric.

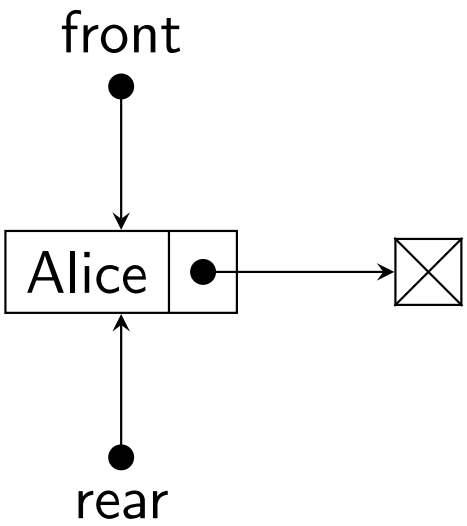


Enqueue: special case

If there is nothing in the queue, the front and rear point to null/None.
So to add Alice, we point front and rear to Alice and set Alice to be the queue.



Enqueue: special case



Enqueue: Pseudo code

Algorithm enqueue

Input: L a linked-list representing a queue, *head* and *tail* two pointers (integer indexes) representing the front and the rear ranks, *elem* an element (object) to add to the queue

Output: none

{First we create a proper node from the element *elem*. Note that the next pointer for a new node is null/None}

$e \leftarrow newNode(elem)$

if L is not *null* **then**

$tail.next() \leftarrow e$

else

$L \leftarrow e$

$head \leftarrow e$

end if

$tail \leftarrow e$

What happens

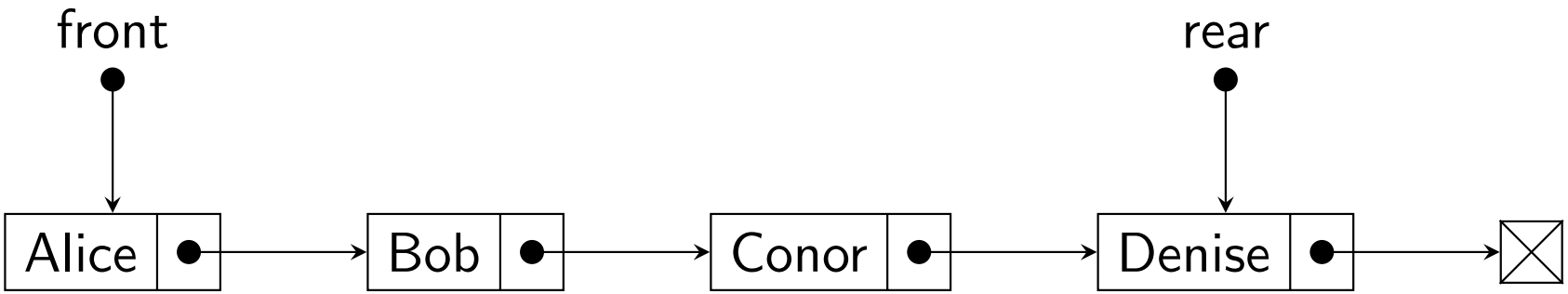
Element e is a node that gets appended to the queue if it has elements.
Otherwise e replaces the empty queue L with node e and points the head and tail to e.

Algorithm enqueue

$e \leftarrow newNode(elem)$
if L is not *null* **then**
  $tail.next() \leftarrow e$
else
  $L \leftarrow e$
  $head \leftarrow e$
end if
$tail \leftarrow e$

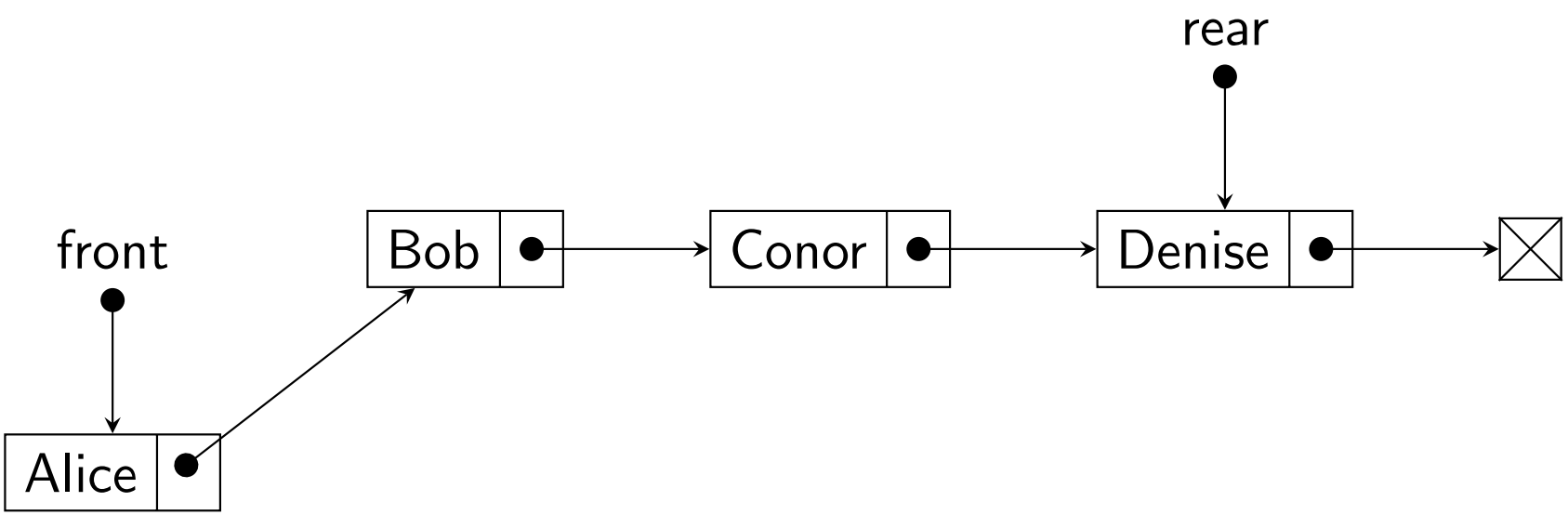
Dequeue

This time Alice, Bob, Conor and Denise are in the queue and we want to remove Alice from the head of the queue.



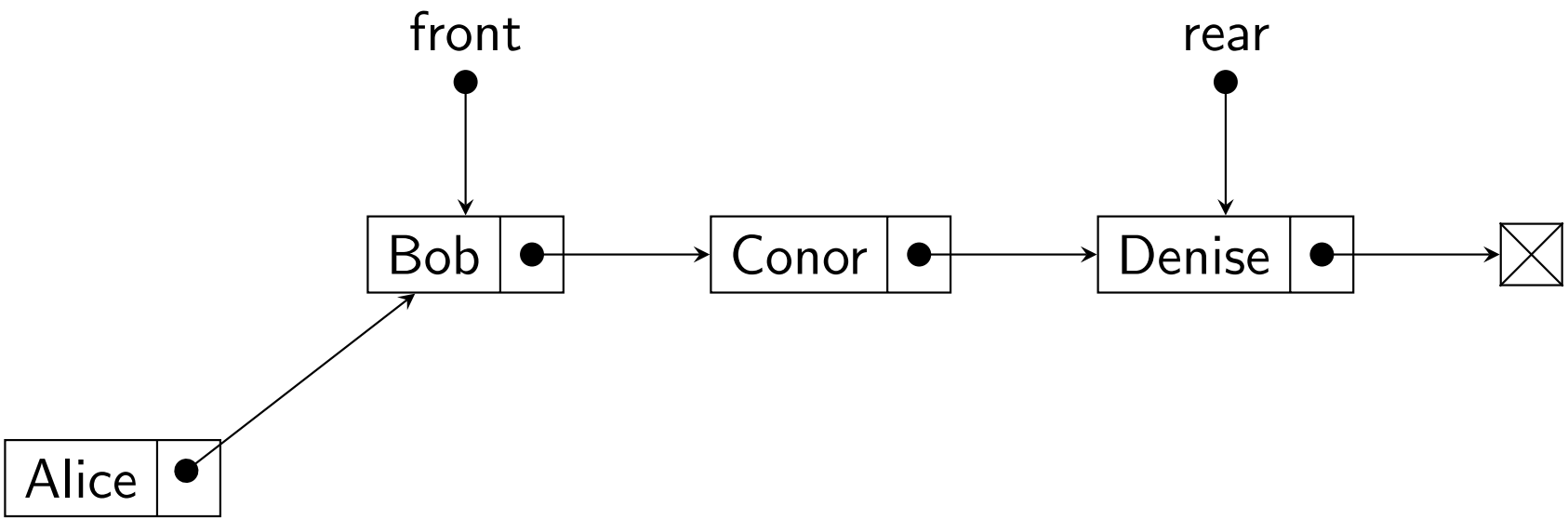
Dequeue

Alice steps out of the queue...



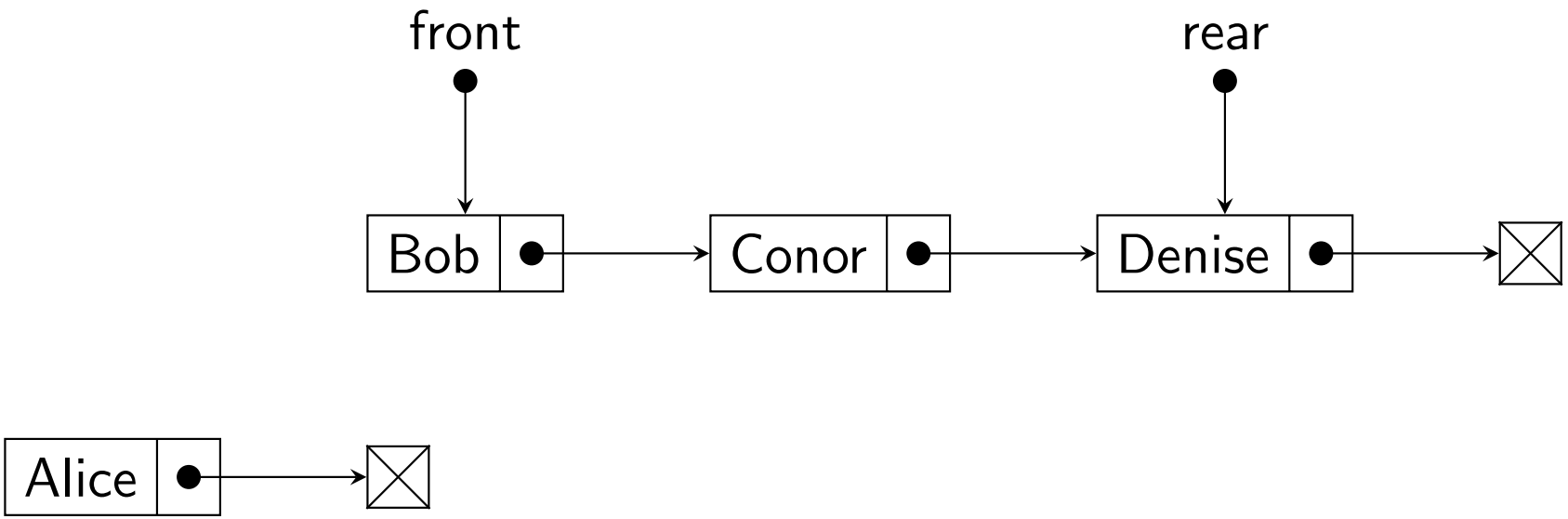
Dequeue

Alice is no longer standing in the queue so the front of the queue is reassigned to Bob.
But Alice is still attached to the queue



Dequeue

So we detach Alice from the queue (element.next set to null/None).
We can now use the detached node as our return value and return the "dequeued" Alice



Dequeue: Pseudo code

Algorithm dequeue

Input: L a linked-list representing a queue, *head* and *tail* two pointers (integer indexes) representing the front and the rear ranks

Output: returns *elem* an element (object) from the head of the queue and removes it from the queue

```
if L is not null then
    elem.element() ← head.element()
    head ← head.next()
    elem.next() ← null
    return elem
else
    error - empty queue
end if
```

What happens

If the queue has elements in it then assign the value from the head of the queue to the *elem* variable so we can return it. Then move the head point to the next node in the list. Finally assign null to the *elem* node's next() address detaching and dereferencing it from the linked list.

Algorithm dequeue

---

if L is not *null* **then**
  $elem \leftarrow head.element()$
  $head \leftarrow head.next()$
  $elem.next() \leftarrow null$
  return elem
else
  error - empty queue
end if

Array Based Queue

Similar to using a linked list?

Do not need to access elements inside the queue
Fast access any array[index] in $\mathcal{O}(1)$ is not valuable
AND... we need to modify the array every time we queue or dequeue an element

Array based enqueue

Add Paul to the tail queue after Jane

Alice	Bob	Conor	Denise	Frank	Jane

Paul

Array based enqueue

We needed to increase the size of the array

Alice	Bob	Conor	Denise	Frank	Jane	Paul

Array based enqueue

Algorithm enqueue

Input: A an array representing a queue of size n, *elem* an element
Output: add an element at the rear of the queue
increase the size of $A(+1)$
$A[n] \leftarrow elem$

Array based dequeue

We want to return Alice and make Bob the new head of the queue

Alice	Bob	Conor	Denise	Frank	Jane	Paul

Array based dequeue

We want to return Alice and make Bob the new head of the queue

	Bob	Conor	Denise	Frank	Jane	Paul

Return:

Alice

Array based dequeue

Reindex/resize array

Array based dequeue

Reindex/resize array

Bob	Conor	Denise	Frank	Jane	Paul

Alice

Array based dequeue

Algorithm dequeue

Input: A an array representing a queue of size n
Output: returns *elem* an element (object) from the head of the queue and removes it from the queue

```
if n > 0 then
    temp ← A[0]
    for i =1 to n -1 do
        A[i-1] ← A[i]
    end for
    decrease the size of A (-1)
    return temp
else
    error - empty queue
end if
```

Analysis of Queue Operations

Operations	Array	List
`size`	$\mathcal{O}(1)$	$\mathcal{O}(n)$ or $\mathcal{O}(1)^*$
`is_empty`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`enqueue`	$\mathcal{O}(n)$	$\mathcal{O}(1)$
`dequeue`	$\mathcal{O}(n)$	$\mathcal{O}(1)$
`first`	$\mathcal{O}(1)$	$\mathcal{O}(1)$

*depends on implementation – whether a node count is stored in queue data structure

Beyond the basics

Double-Ended Queues or "Deques" (Decks)

A queue-like data structure that supports insertion and deletion at **both the front and the back** of the queue.
Pronounced âdeckâ to avoid confusion with the dequeue ADT operation!

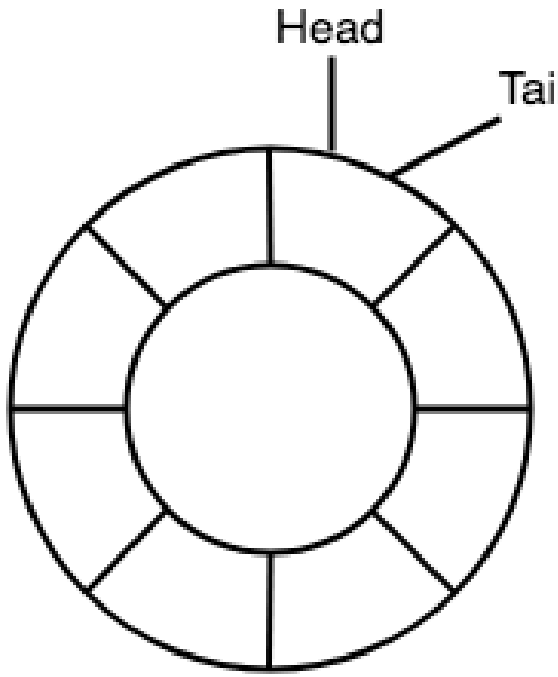
Circular Array Queue Implementations

The contents of the queue to âwrap aroundâ the end of an underlying array – join the beginning to the end of the array.
Assume underlying array has fixed length
$N > numberofelementsinthequeue$.

Circular Queue: Initial State

Imagine the memory is wrapped around and the start and end of the array are joined together looping and closing like a belt buckle.

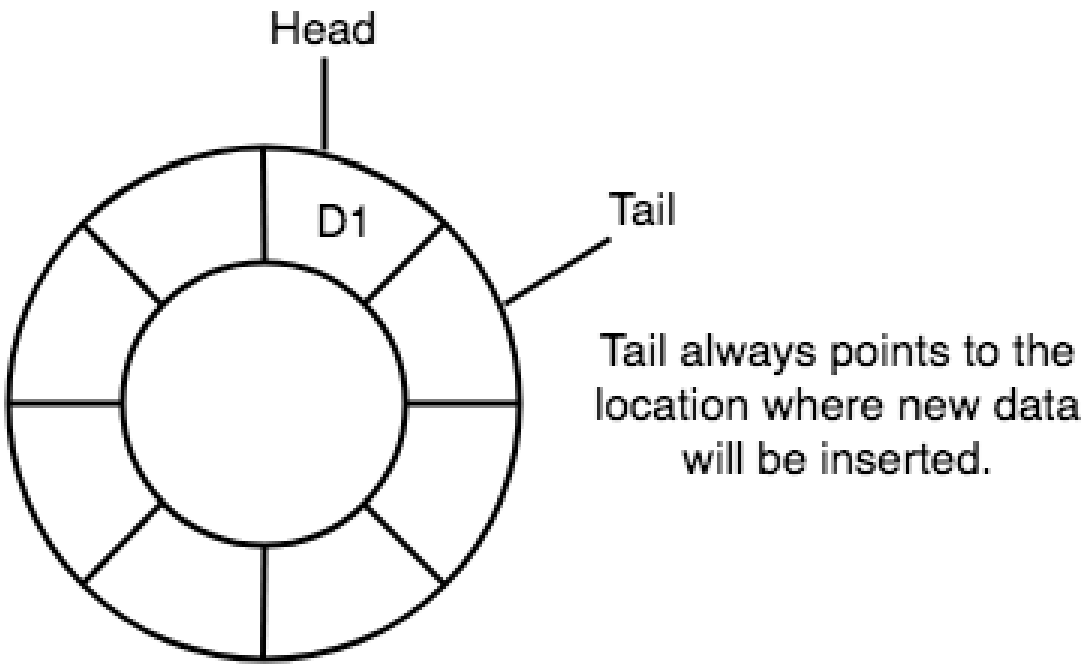
Initially the queue is empty, as Head and Tail are at same location



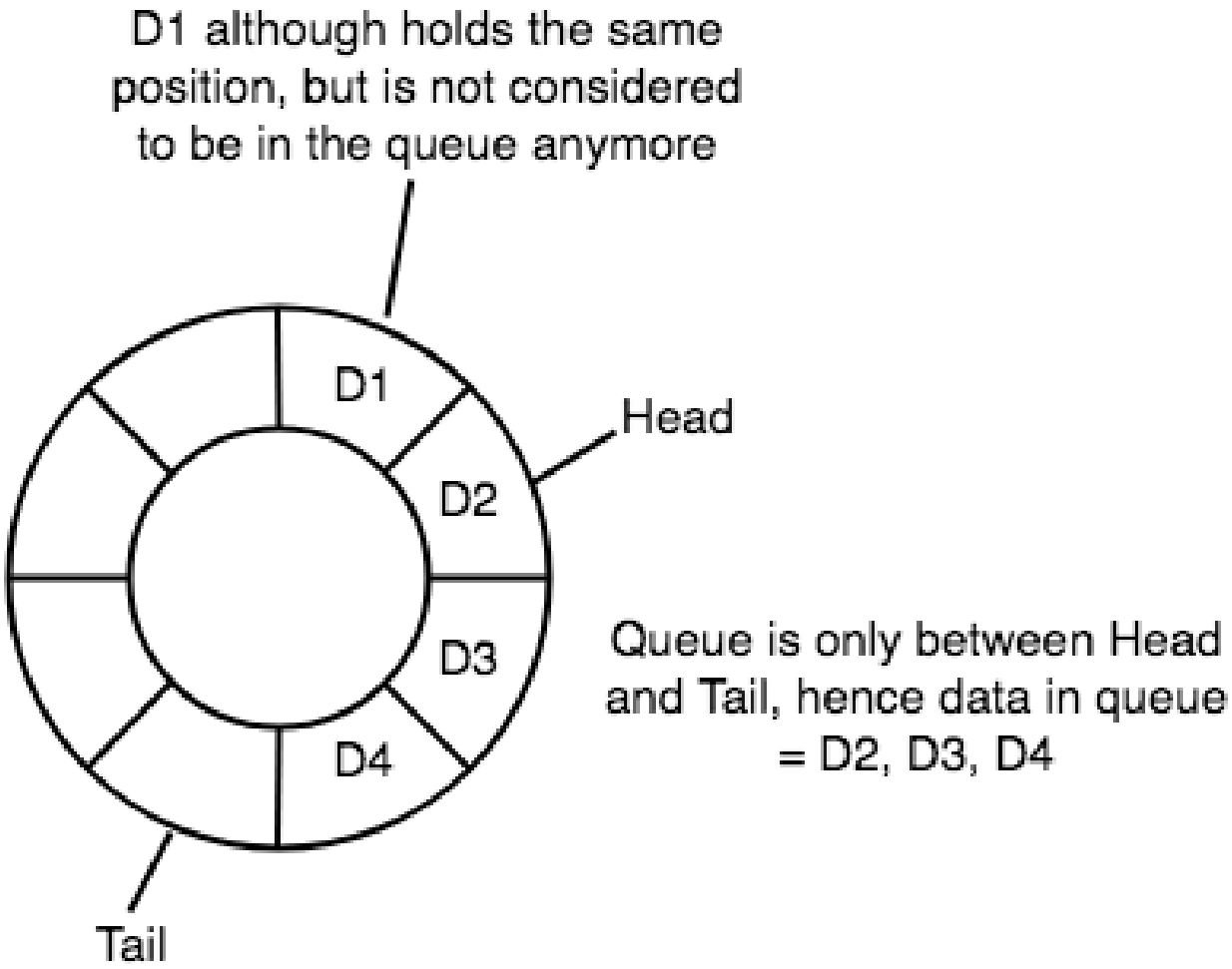
A simple circular queue with size 8

Image Source: `https://www.studytonight.com/data-structures/circular-queue`

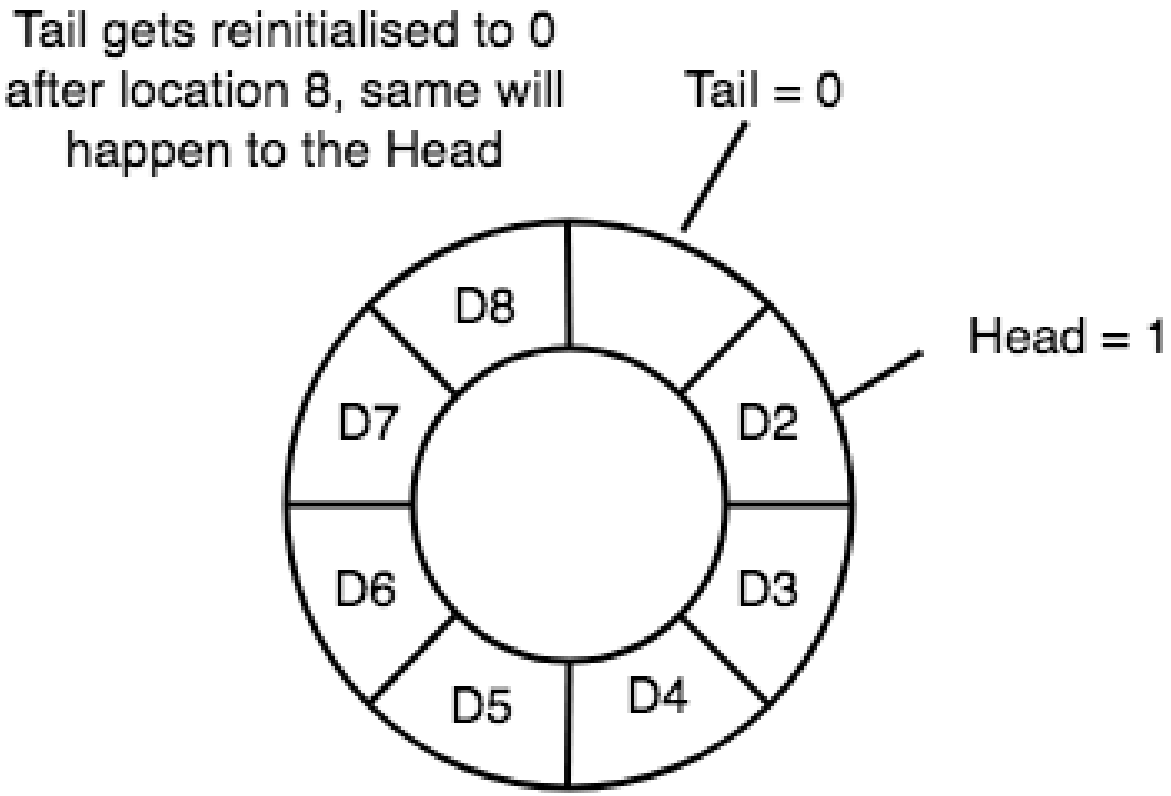
Circular Queue: Add Data



Circular Queue: Remove Data



Circular Queue: Reinitialise



Round and Round: Remainder

```
head = (head+1) % maxSize
tail = (tail+1) % maxSize
```