## Queues

1. https://www.coursera.org/learn/data-structures/lecture/EShpq/queues

**Queue**:

* ***FIFO*** (*First In First Out*) data structure (abstract data type).
* Each queue operation is **O(1)**: Enqueue, Dequeue, Empty.
* Can be implemented with either a linked list (with tail pointer) or an array (with a read and a write indices).
  + An array implementation:
    - Has a maximum size that the queue can grow to.
    - Any unused indices are wasted space.
  + A linked list implementation:
    - Can get as large as needed as long as there’s available memory.
    - More space is needed, because we have to allocate space for both the key and the pointer.
* Example: waiting in line at the supermarket.

It has the following operations.

* **Enqueue(Key)**: adds key to collection
* **Key Dequeue()**: removes and returns least recently-added key
* **Boolean Empty()**: are there any elements?

### Queue Implementation with Linked-List.

Operations:

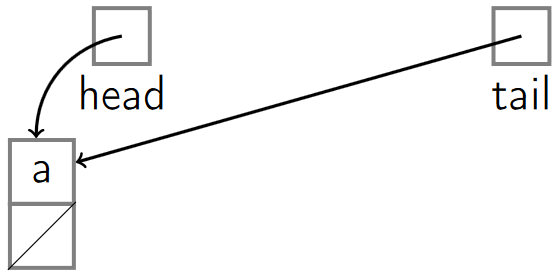
* **Enqueue**: uses *List.PushBack*
* **Dequeue**: uses *List.TopFront* and *List.PopFront*
* **Empty**: uses *List.Empty*

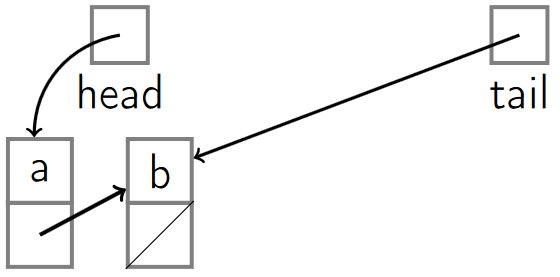
#### Enqueue O(1)

1. Push to the back of the linked list (uses *List.PushBack*)



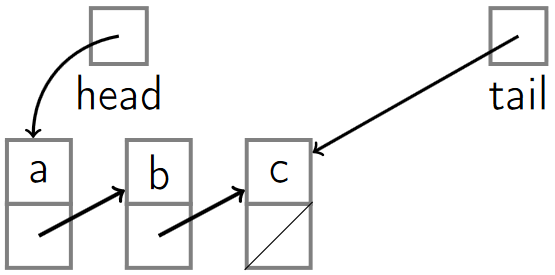
2. If there’re no elements in the queue, both the tail and the head’s pointer are set to the newly enqueued node.

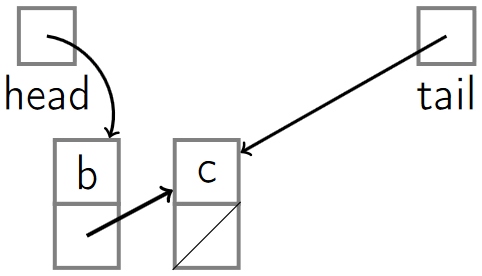




#### Dequeue O(1)

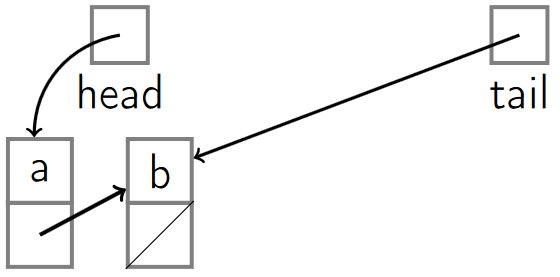
1. Removes the first element of the list, where the head points to (uses *List.TopFront* and *List.PopFront*).





#### Empty O(1)

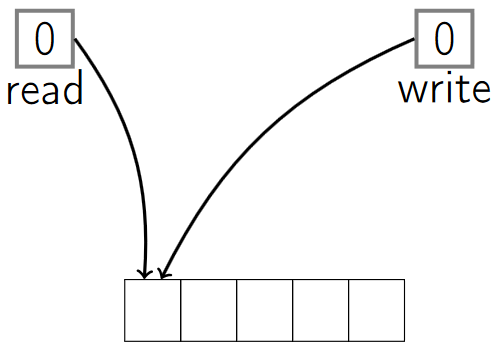
1. Check if the head and tail both point to *null* (uses *List.Empty*).





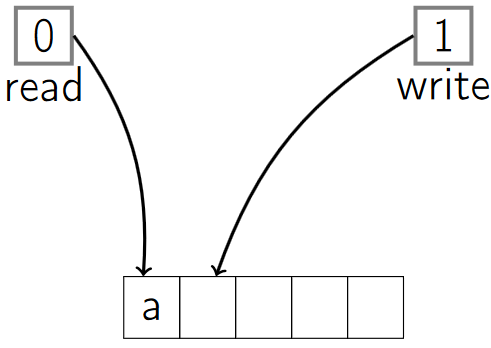
### Queue Implementation with Array.

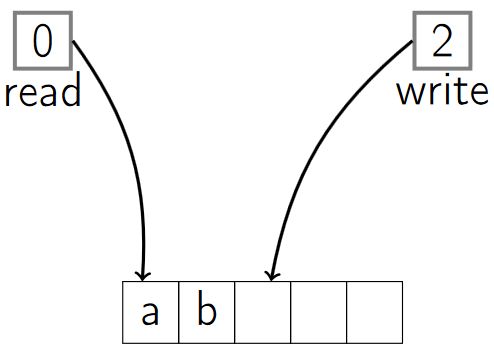
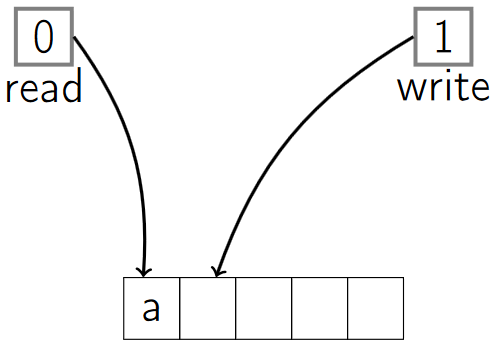
We need to keep track of the array as a *circular array* using a *read and write indices*.

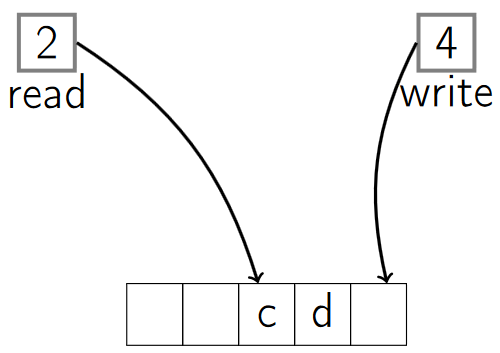
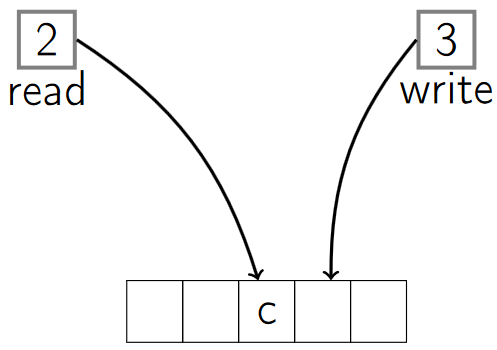


#### Enqueue O(1)

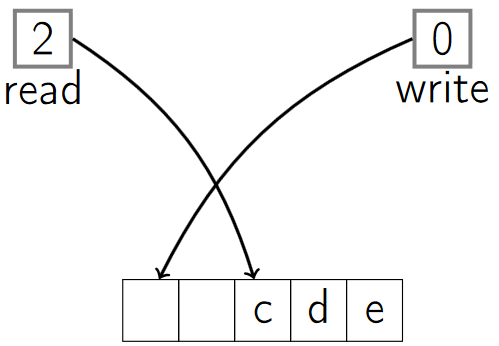
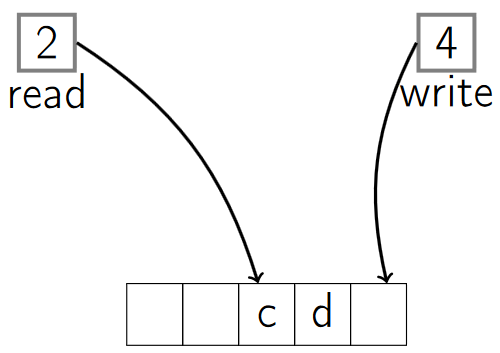
1. The *write index* tells us where the next *Enqueue* operation should happen.
2. When *enqueuing*, we read what’s at the *write index* and then we increment the *write index* by 1.



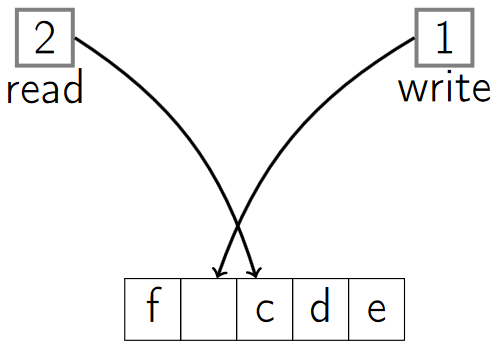




1. When the *write index* reaches the length of the array, we restart it to the first index (0).

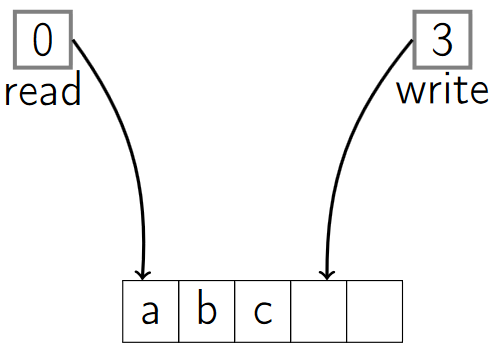


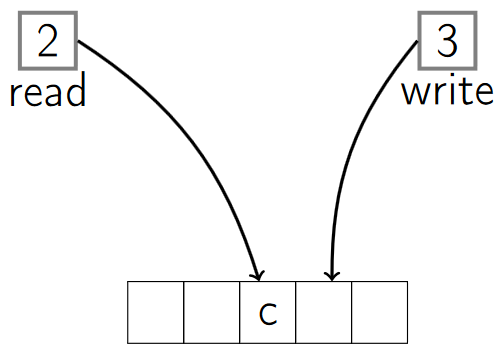
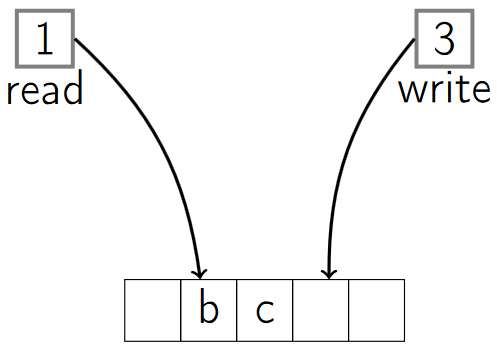
1. The *read* and *write indices* can’t be both the same number (we have a buffer of at least one element that can’t be written to, to make sure read and write are separate and distinct if the queue’s not empty).



#### Dequeue O(1)

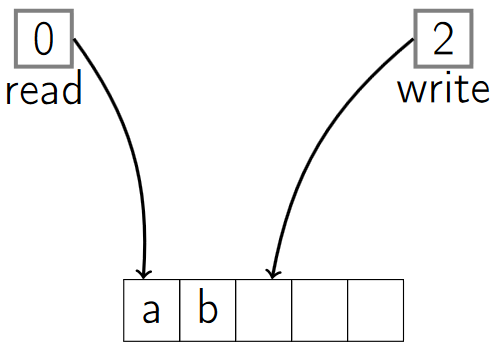
1. Expensive operation, we would need to move the elements any time the top is removed.
2. The *read index* tells us where the next *Dequeue* operation should happen.
3. When *dequeueing*, we read what’s at the *read index* and then increment the *read index* by 1.

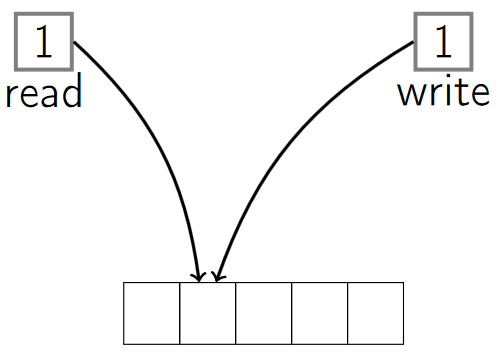




#### Empty O(1)

1. We check whether the *write index* is equal to the *read index* , if not then there’s something to *dequeue* that has been *enqueued*.





# Trees

## Walking a tree