# CS 225 - Lecture 8

Scribe : Harsha Srimath Tirumala

## 1 Learning Goals

↪ Queue : FIFO data structure

↪ Queue Implementation

↪ Circular Queue

↪ Iterators : motivation, definition

## 2 Queue - First In First Out

↪ A queue stores an ordered collection of items. Queues support only two operations - *Enqueue* and *Dequeue* :

- *Enqueue* : Put an item at the *back* of the queue.
- *Dequeue* : Remove and return the *front* item of the queue.

Since elements can only be inserted at the back and removed from the front, queues follow the First In First Out model. Queues do NOT support random access.

### 2.1 Implementation

↪ Requirements : *Enqueue* - $O(1)$ InsertBack and *Dequeue* - $O(1)$ RemoveFront.

↪ A *Linked list* already supports $O(1)$ RemoveFront via the *head_* pointer. In order to support $O(1)$ InsertBack, storing an additional *tail_* pointer suffices.

↪ C++ implementation - Using a vector/deque (which themselves use an *array list*). Why?

1. Engineering - In practice, repeated calls for new allocation can be expensive. An array bypasses this due to having a continuous chunks of memory; although,in CS 225 we will ignore these issues.
2. We can improve theory!

## 3 Queues using array lists

↪ Queues require $O(1)$ InsertBack and $O(1)$ RemoveFront. Array lists do support $O(1)$ InsertBack - as long as the list is *not at capacity*. We now need to figure out how to implement RemoveFront in $O(1)$.

↪ Deisng choice - For the queue implementation, *cap, size, front* will all be of type *unsigned int*.

### 3.1 RemoveFront in $O(1)$ time using array lists

↪ Declaring a new variable *front* and using it to point to the element at the front can support RemoveFront in $O(1)$ time. *front* should be updated every time there is a dequeue operation.

### 3.2 Circular queue - motivation

↪ Consider a queue with $n$ enqueue operations followed by $(n-3)$ dequeue operations. This queue has $(n-3)$ available slots. Based on the implementation of *front* as discussed above, the queue has **no space** to enqueue any more elements (as *front* is $(n-3)$ and the last three elements occupy positions $(n-3)$ thru $(n-1)$ - which is the last position). Such a waste!

↪ In order to reuse spaces left unoccupied, we implement Queues in a circular fashion using array lists. This allows us to reclaim these spaces and utilize memory efficiently.

## 4 Circular Queue data structure

For the circular implementation, we need to manipulate variables to go around the "end" of the array list.

↪ Enqueue($data$) : if($cap! = size$):

1. InsertBack : Insert *data* at index $((front + size)\%cap)$
2. Update *size* : $size{+}{+}$

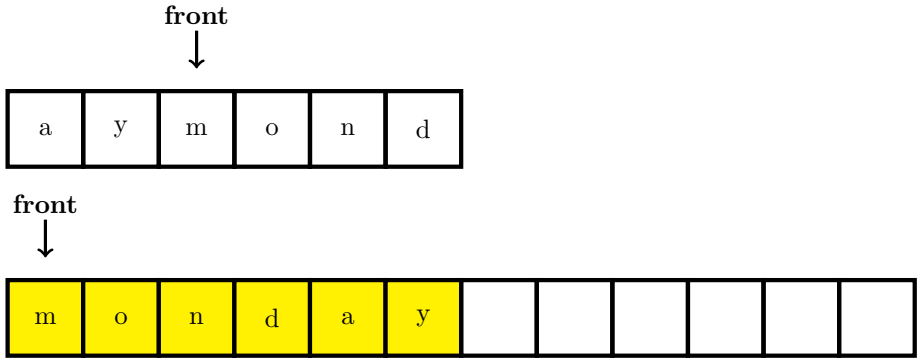
↪ Dequeue() : if($size! = 0$):

1. RemoveFront : Remove data from index *front*
2. Update *front* : $front = (front + 1)\%cap$

### 4.1 Circular Queue : Resizing

We have seen above that *enqueue* works as long as the queue is not at capacity. How to insert at capacity? Well, we follow the Resize x2 strategy that was used for array lists.

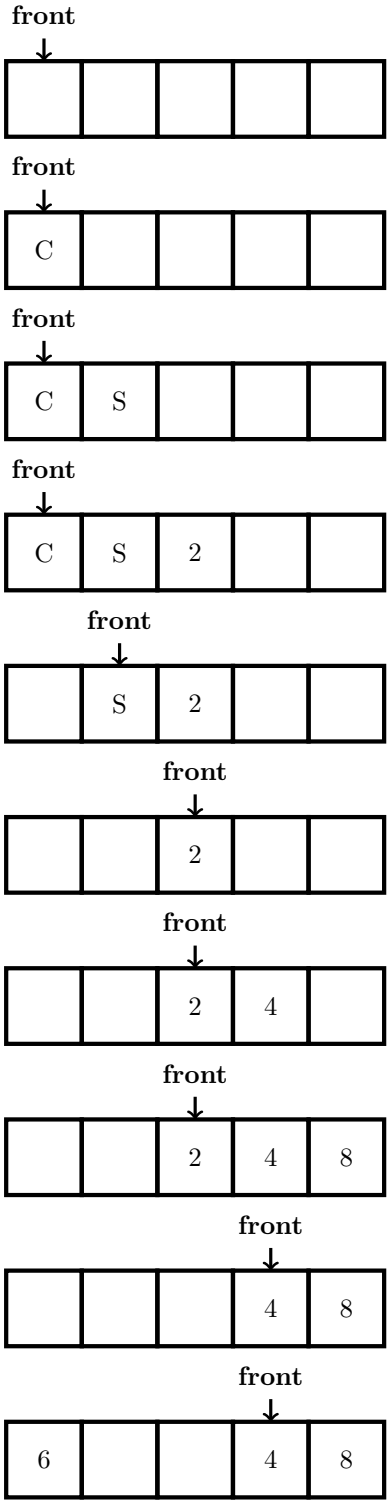
While copying contents onto the new array list, it is critical to ensure that *front* gets copied to the first slot of the new list followed by successive elements as shown below. This ensures localization/continuity of successive elements (particularly after subsequent *enqueue* operations).



## 4.2 Circular Queue : Simulation

Simulation of a queue Q with the following sequence of operations -

a) *enqueue*(C) b) *enqueue*(S) c) *enqueue*(2) d) *dequeue*() e) *dequeue*()
f) *enqueue*(4) g) *enqueue*(8) h) *dequeue*() i) *enqueue*(6)



### 4.3 Queue vs Stack

Table 1: Queue vs Stack

| | Order | Implementation | Insert/Remove | Random Access |
|---|---|---|---|---|
| Queue | FIFO | "Circular" Array list | $O(1)^*$ | X |
| Stack | LIFO | Linked list | $O(1)$ | X |

Note : Queues support amortized $O(1)$ for *enqueue*.

## 5 Iterators

↪ Motivation - We have seen how to iterate through array lists or linked lists to access different items stored in a given list. How do we go about this task if our objects are not "standard"? (for example : vertices of a cube)

↪ Iterators - Iterators provide a systematic way to access items in a container without exposing the underlying structure of the container.

↪ Requirements - For a class to have an iterator, it needs two functions :

- Iterator begin() : first item
- Iterator end() : last item

↪ The actual iterator is defined as a class *inside* the outer class:

1. It must be of base class **std::iterator**.
2. It must implement at least the following operations :
   - ↪ GetNext : Iterator& operator ++()
   - ↪ Dereference : Const T& operator* ()
   - ↪ Compare Objects : bool operator !=(const Iterator&)