

04-630

Data Structures and Algorithms for Engineers

David Vernon
Carnegie Mellon University Africa

vernon@cmu.edu
www.vernon.eu

Lecture 11

Queue ADT

Implementation using List ADT (array and linked-list)

Comparison of order of complexity

Dedicated ADT

Circular queues

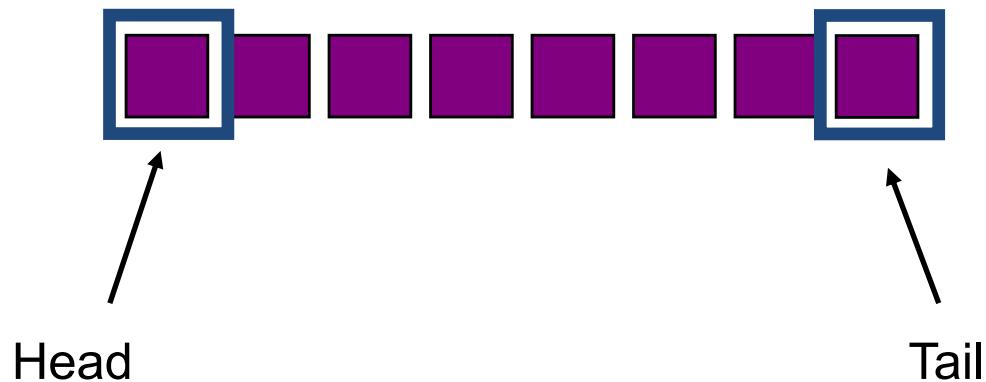
Queue applications

Queues

A queue is another special type of list

- insertions are made at one end, called the tail of the queue
- deletions take place at the other end, called the head
- thus, the last one added is always the last one available for deletion
- also referred to as
 - FiFO list (First In First Out)

Queues



Queue Operations

Declare. $\rightarrow Q :$

The function value of $Declare(Q)$ is an empty queue

Queue Operations

Empty. → **Q** :

The function *Empty* causes the queue to be emptied and it returns position *End(Q)*



Queue Operations

IsEmpty. $\mathbf{Q} \rightarrow \mathbf{B}$:

The function value $\text{IsEmpty}(Q)$ is *true* if Q is empty; otherwise it is *false*

Queue Operations

Head: $Q \rightarrow E$:

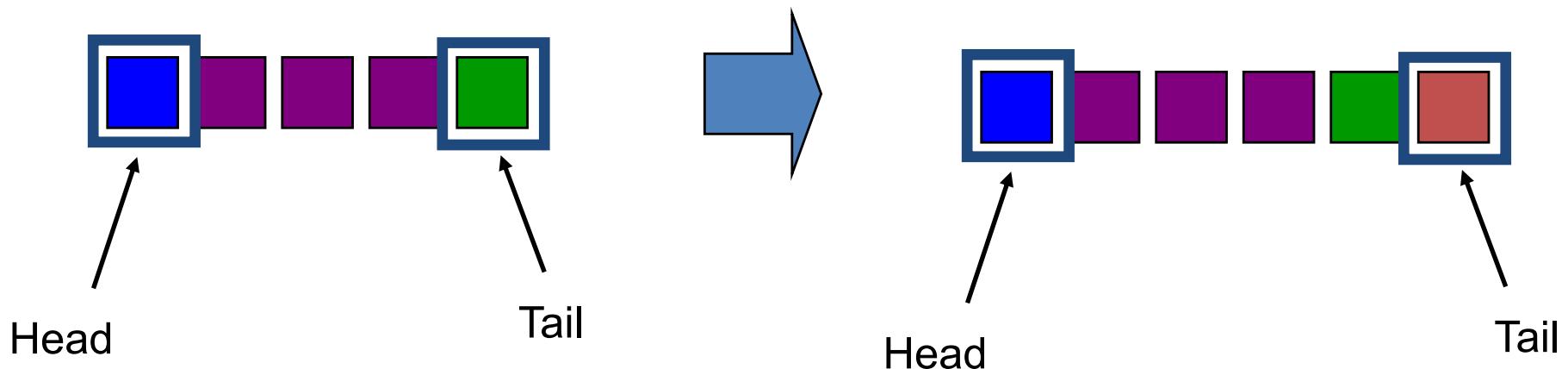
The function value $Head(Q)$ is the first element in the list;
if the queue is empty, the value is undefined

Queue Operations

Enqueue: $E \times Q \rightarrow Q$:

Enqueue(e, Q)

Add an element e to the tail of the queue

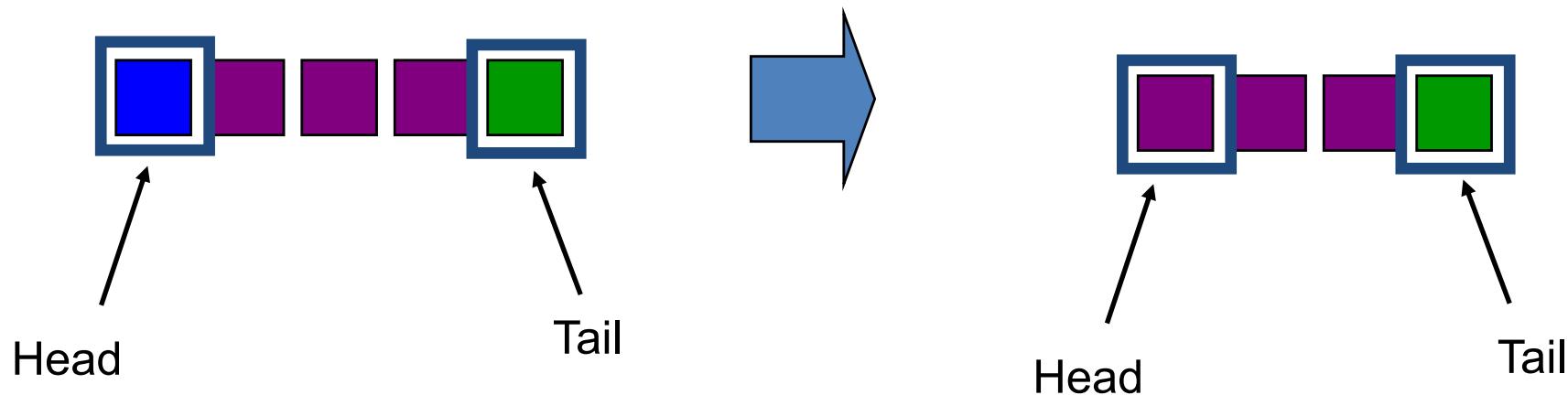


Queue Operations

Dequeue: $Q \rightarrow E$:

Dequeue(Q)

Remove the element from the head of the queue: i.e. return the first element and delete it from the queue



Queue Operations

- All these operations can be directly implemented using the LIST ADT operations on a queue Q
- Again, it may be more efficient to use a dedicated implementation
- And, again, it depends what you want: code efficiency or software re-use (i.e. utilization efficiency)

Queue Operations

Declare(Q)

Empty(Q)

Head(Q)

 Retrieve(First(Q), Q)

Enqueue(e, Q)

 Insert(e, End(Q), Q)

Dequeue(Q)

 Retrieve(First(Q), Q)

 Delete(First(Q), Q)

Queue Errors

- Queue **overflow** errors occur when you attempt to **enqueue()** an element in a queue that is **full**
- Queue **underflow** errors occur when you attempt to **dequeue()** an element from an empty queue
- Your ADT implementation should provide guards that catch these errors

Queue Implementation

- The List ADT can be implemented
 - As an array
 - As a linked-list
- So, therefore, so can the Queue ADT
- What are the advantages and disadvantages of these two options?
- When would you pick one implementation over the other?

Queue Operations

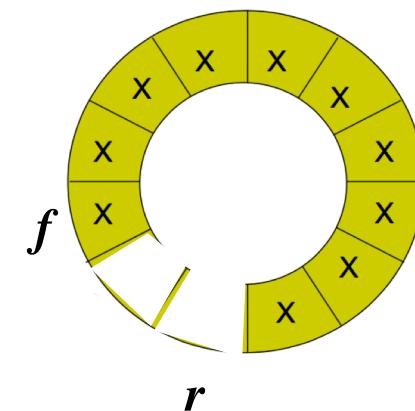
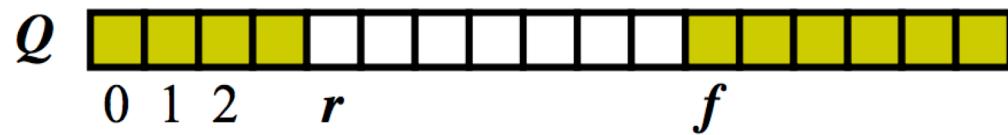
	Array	Linked-List
Declare(Q)	$O(1)$	$O(1)$
Empty(Q)	$O(1)$	$O(n)$
Head(Q)	$O(1)$	$O(1)$
Retrieve(First(Q), Q)		
Enqueue(e , Q)	$O(1)$	$O(n) \dots \text{why?}$
Insert(e , End(Q), Q)		
Dequeue(Q)	$O(n) \dots \text{why?}$	$O(1)$
Retrieve(First(Q), Q)		
Delete(First(Q), Q)		

Queue Implementation

- Reusing the List ADT involves some compromises
- Alternative is to create a new Queue ADT
 - With an implementation that avoids these compromises

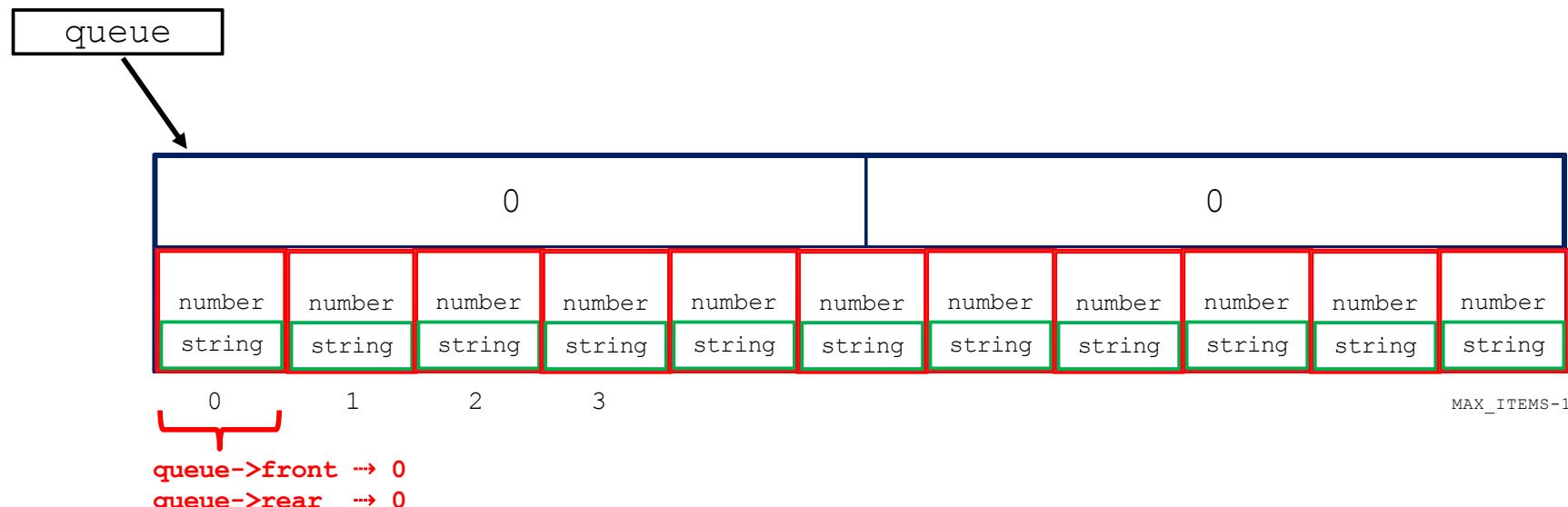
Queue Implementation

```
typedef struct {
    int front;
    int rear;
    ITEM_TYPE items[MAX_ITEMS];
} QUEUE_TYPE;
```



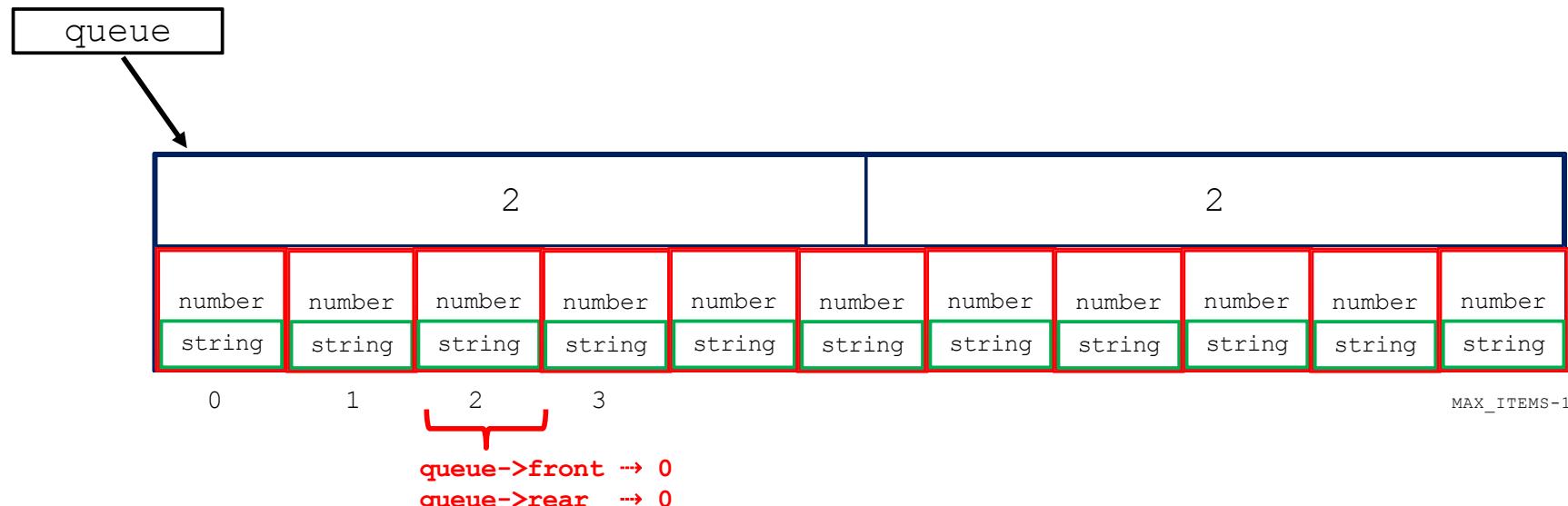
Queue Implementation

```
void empty(QUEUE_TYPE *queue) {  
    queue->front = 0;  
    queue->rear = 0;  
    return(end(queue));  
}
```



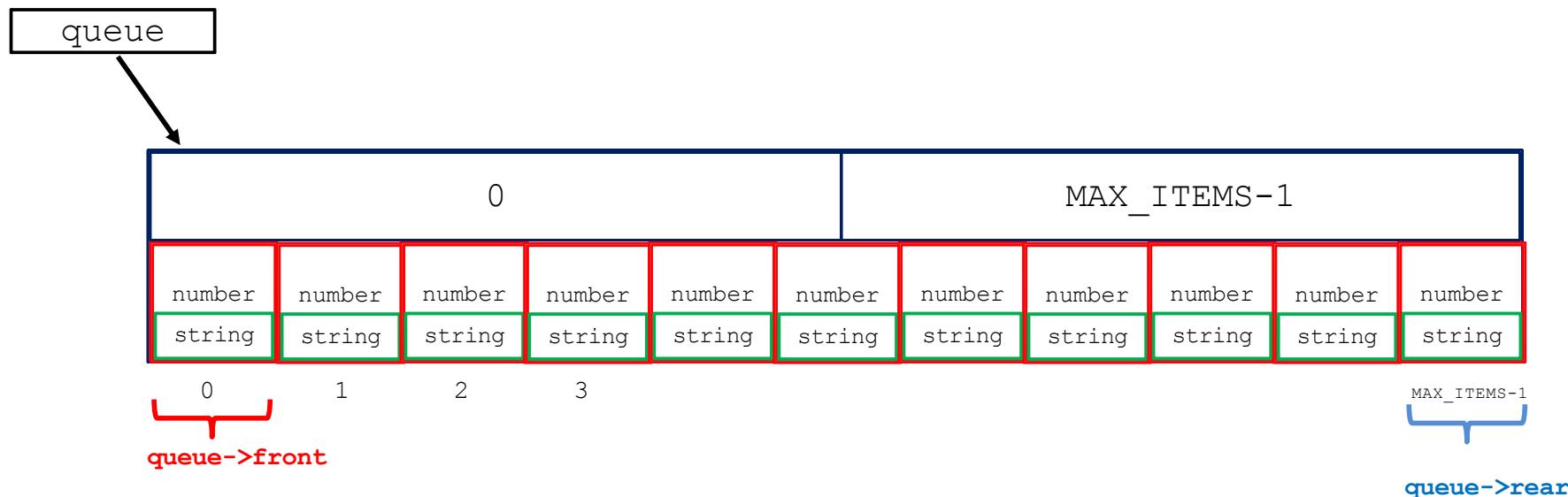
Queue Implementation

```
bool is_empty(QUEUE_TYPE *queue) {  
    if (queue->front == queue->rear)  
        return(true);  
    else  
        return(false)  
}
```



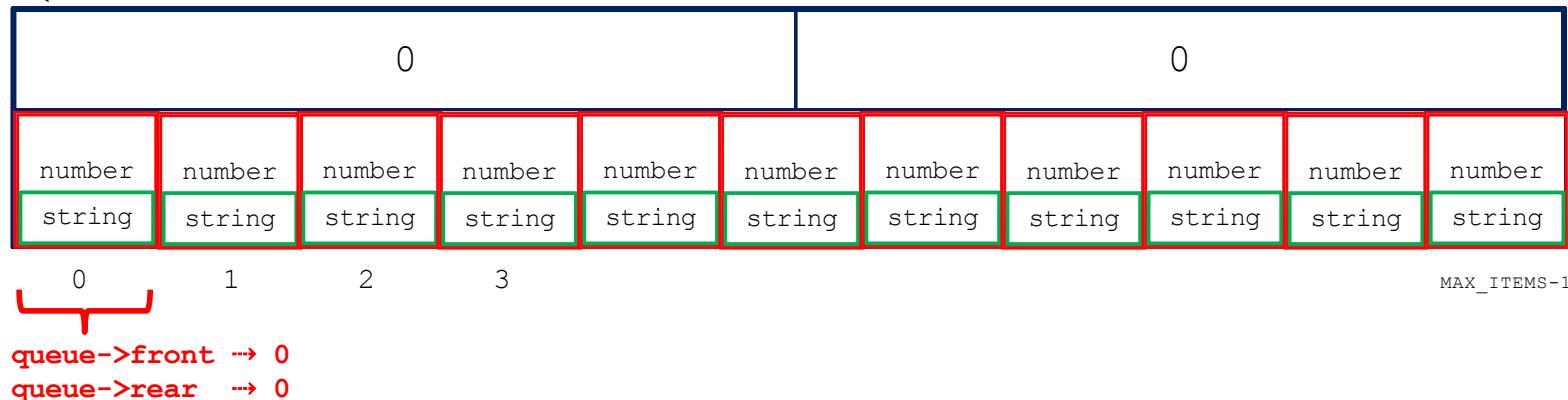
Queue Implementation

```
int is_full(QUEUE_TYPE *queue) {
    if (( rear + 1 ) % MAX_ITEMS == front )
        return(TRUE);
    else
        return(FALSE);
}
```



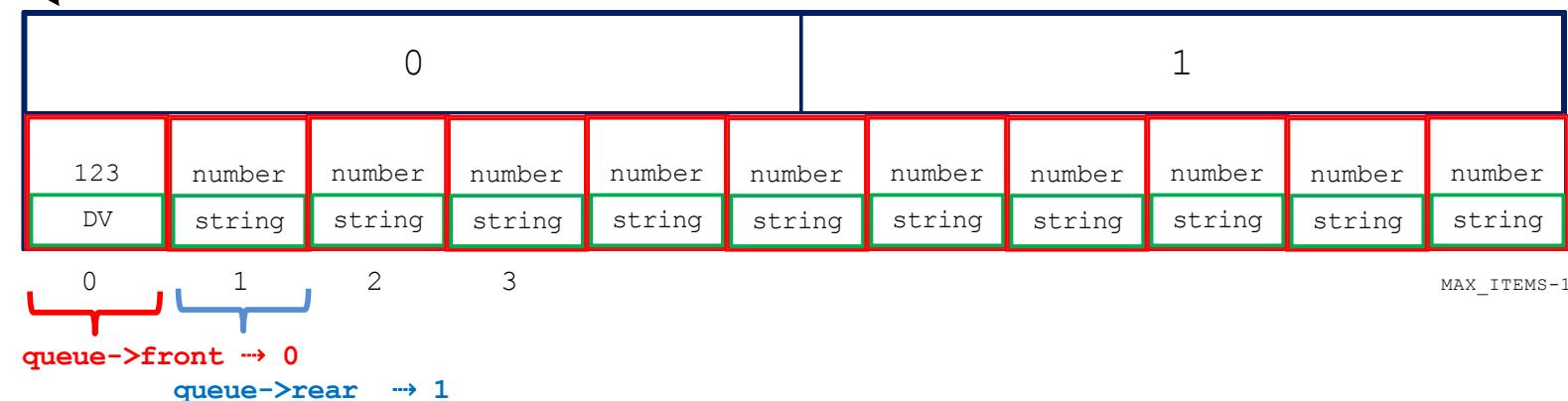
Queue Implementation

```
void enqueue(ITEM_TYPE e, QUEUE_TYPE *queue) {  
    if (!is_full(queue)) {  
        items[rear] = e;  
        rear = (rear +1) % MAX_ITEMS;  
    }  
    else {  
        error("Queue overflow: queue is already full");  
    }  
}
```



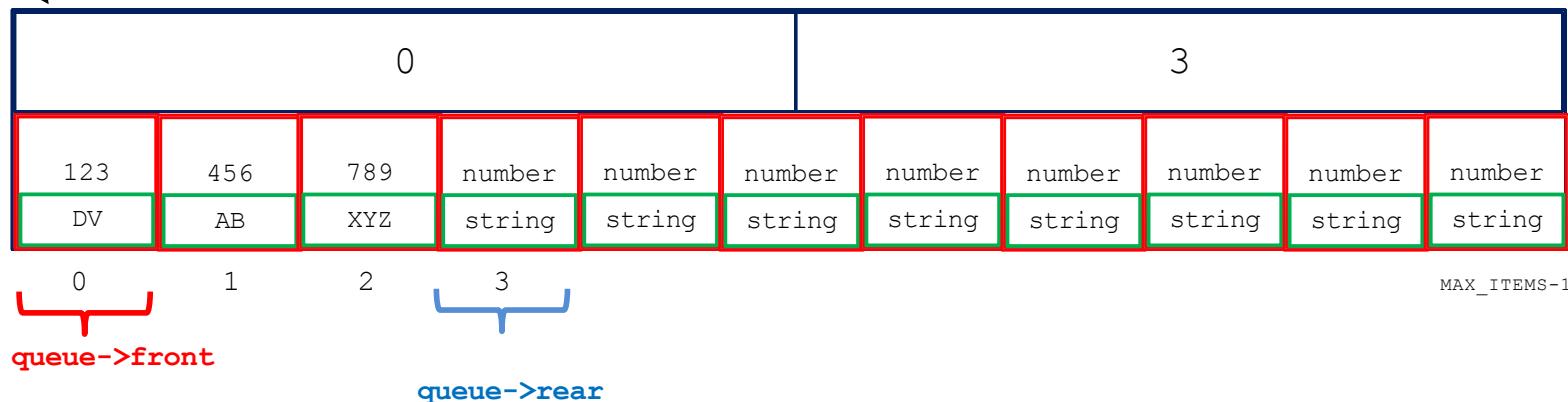
Queue Implementation

```
void enqueue(ITEM_TYPE e, QUEUE_TYPE *queue) {  
    if (!is_full(queue)) {  
        items[rear] = e;  
        rear = (rear +1) % MAX_ITEMS;  
    }  
    else {  
        error("Queue overflow: queue is already full");  
    }  
}
```



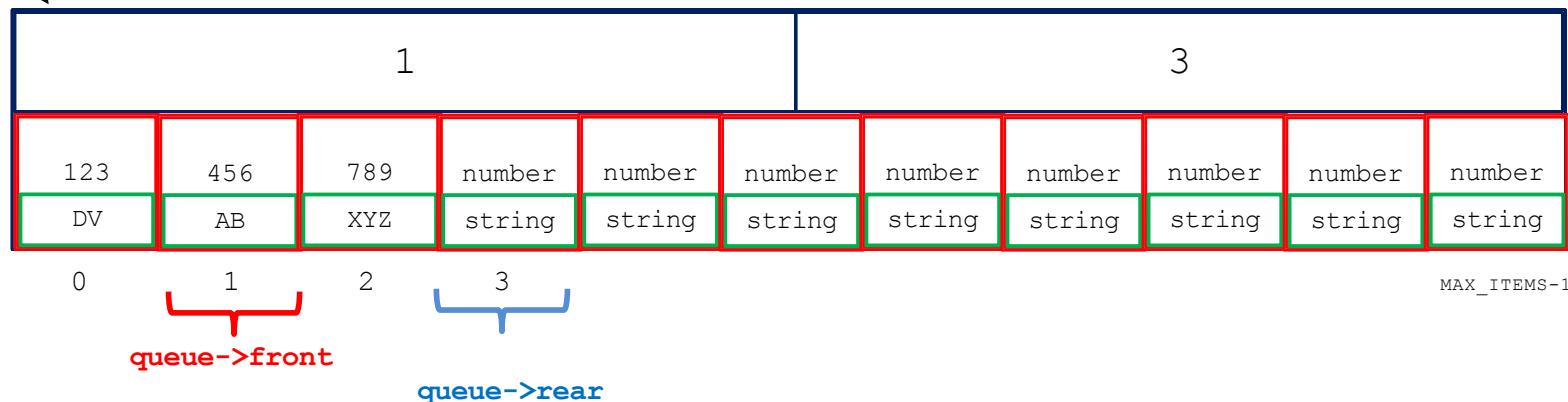
Queue Implementation

```
void dequeue(ITEM_TYPE *e, QUEUE_TYPE *queue) {  
    if (!is_empty(queue)) {  
        *e = items[front];  
        front = (front+1) % MAX_ITEMS;  
    }  
    else {  
        error("Queue underflow: queue is empty");  
    }  
}
```



Queue Implementation

```
void dequeue(ITEM_TYPE *e, QUEUE_TYPE *queue) {  
    if (!is_empty(queue)) {  
        *e = items[front];  
        front = (front+1) % MAX_ITEMS;  
    }  
    else {  
        error("Queue underflow: queue is empty");  
    }  
}
```



Queue Implementation

- Can you see a particular problem with the linked-list implementation?
- How would you fix it?

Queue Operations

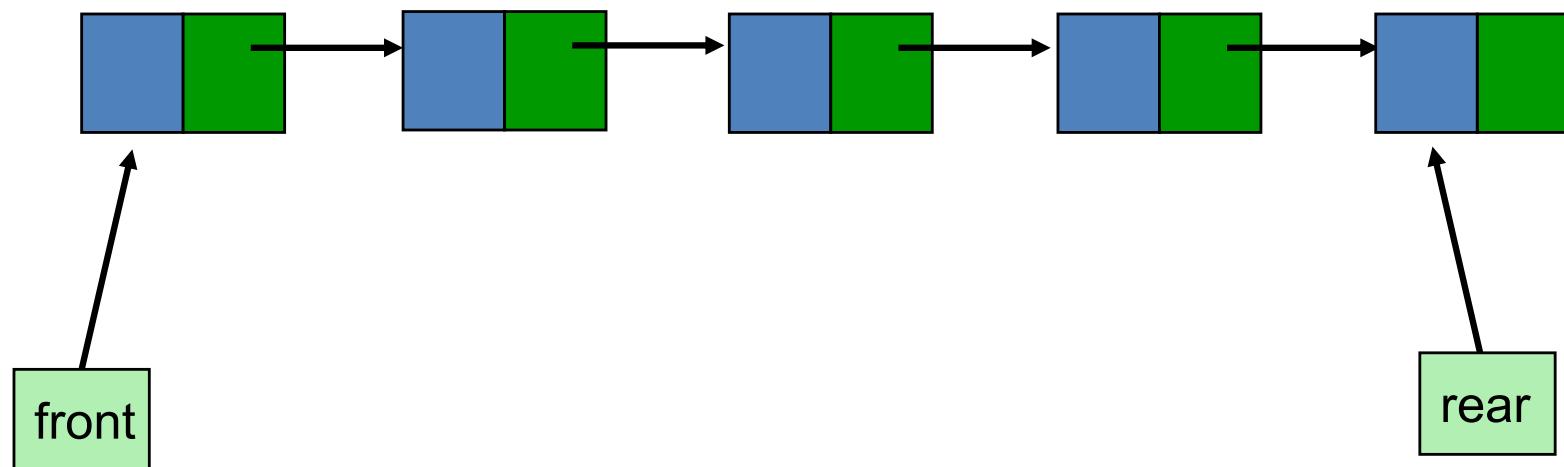
	Array	Linked-List
Declare(Q)	$O(1)$	$O(1)$
Empty(Q)	$O(1)$	$O(n)$
Head(Q)	$O(1)$	$O(1)$
Retrieve(First(Q), Q)		
Enqueue(e , Q)	$O(1)$	$O(n) \dots \text{why?}$
Insert(e , End(Q), Q)		
Dequeue(Q)	$O(n) \dots \text{why?}$	$O(1)$
Retrieve(First(Q), Q)		
Delete(First(Q), Q)		

Queue Implementation

- Can you see a particular problem with the linked-list implementation?
- How would you fix it?

Queue Implementation

- Can you see a particular problem with the linked-list implementation?
- How would you fix it?



Queue Applications

- Scheduling/waiting for system service queues
- Resource queues – provide coordinated access to shared resources
- Message queues (Buffer)
- Multi-queues and priority queues