Unit – I : Elementary Data Structures

- Data structures
- Asymptotic complexity
- Abstract data type:
  - Array,
  - Stacks,
  - **Queues,**
  - Linked Lists, and their applications.

Todays Agenda

- Queue
- Queue Operations
- Summary

Practical Applications - Queue

- Breadth-first search (BFS) algorithm.
- It is used to handle interrupts in real-time systems.
- Job Scheduling, to maintain a queue of processes in operating systems (FIFO order).
- The queue of packets in data communication.
- CPU scheduling, Disk Scheduling.
- When data is transferred asynchronously between two processes, the queue is used for synchronization. For example IO Buffers, pipes, file IO, etc.

Practical Applications - Queue

- Call Center phone systems use Queues to hold people calling them in order.
- Handling website traffic.
- Routers and switches in networking.
- Maintaining the playlist in media players.
- +100..................

Background

- Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue).

- Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

Background

- Queue can handle multiple data.
- We can access both ends.
- They are fast and flexible.

- Like stacks, Queues can also be represented in an array: In this representation, the Queue is implemented using the array. Variables used in this case are
  - Queue: the name of the array storing queue elements.
  - Front: the index where the first element is stored in the array representing the queue.
  - Rear: the index where the last element is stored in an array representing the queue.

Implementation of Queue:

- Sequential allocation: A queue can be implemented using an array. It can organize a limited number of elements.

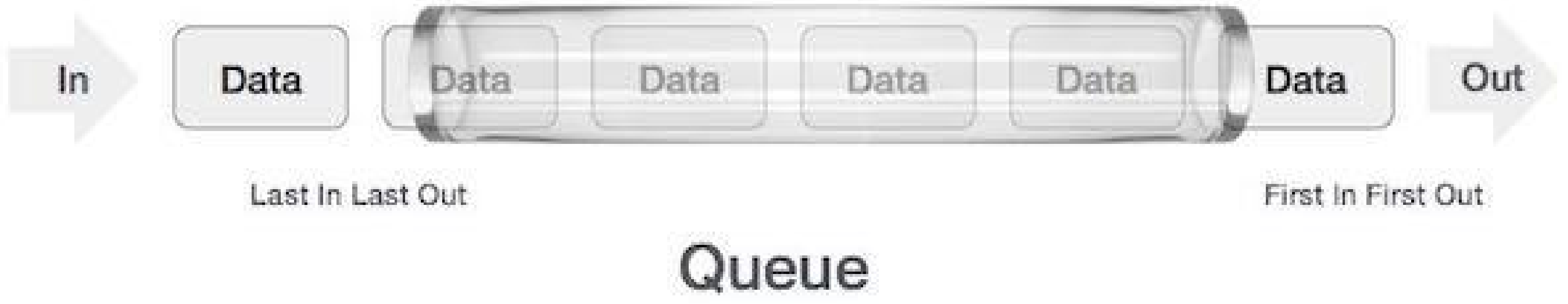
- Linked list allocation: A queue can be implemented using a linked list. It can organize an unlimited number of elements.

Types of Queues

- Simple Queue:
- Circular Queue
- Priority Queue
- Dequeue: Dequeue is also known as Double Ended Queue. As the name suggests double ended, it means that an element can be inserted or removed from both the ends of the queue unlike the other queues in which it can be done only from one end. Because of this property it may not obey the First In First Out property.

Queue Representation

- e access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure −



- As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.

Basic Operations

- Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory.
- The basic operations associated with queues
- **enqueue()** − add (store) an item to the queue.
- **dequeue()** − remove (access) an item from the queue.

Basic Operations

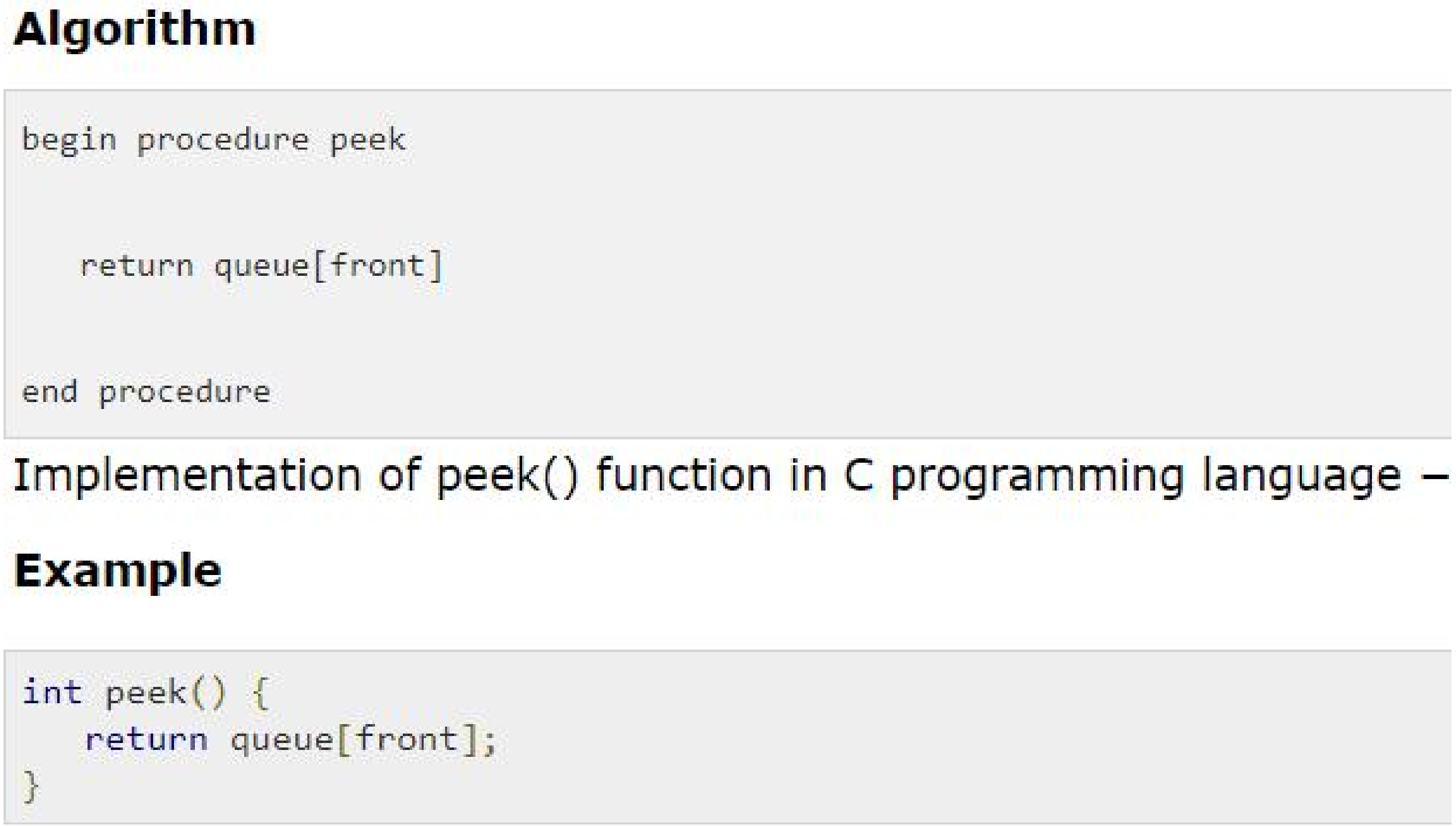
- Few more functions are required to make the above-mentioned queue operation efficient.
- These are −
- **peek()** − Gets the element at the front of the queue without removing it.
- **isfull()** − Checks if the queue is full.
- **isempty()** − Checks if the queue is empty.

Rule

- In queue, we always dequeue (or access) data, pointed by **front** pointer and while enqueing (or storing) data in the queue we take help of **rear** pointer.

peek()

- This function helps to see the data at the **front** of the queue. The algorithm of peek() function is as follows −



Algorithm

```
begin procedure peek

   return queue[front]

end procedure
```

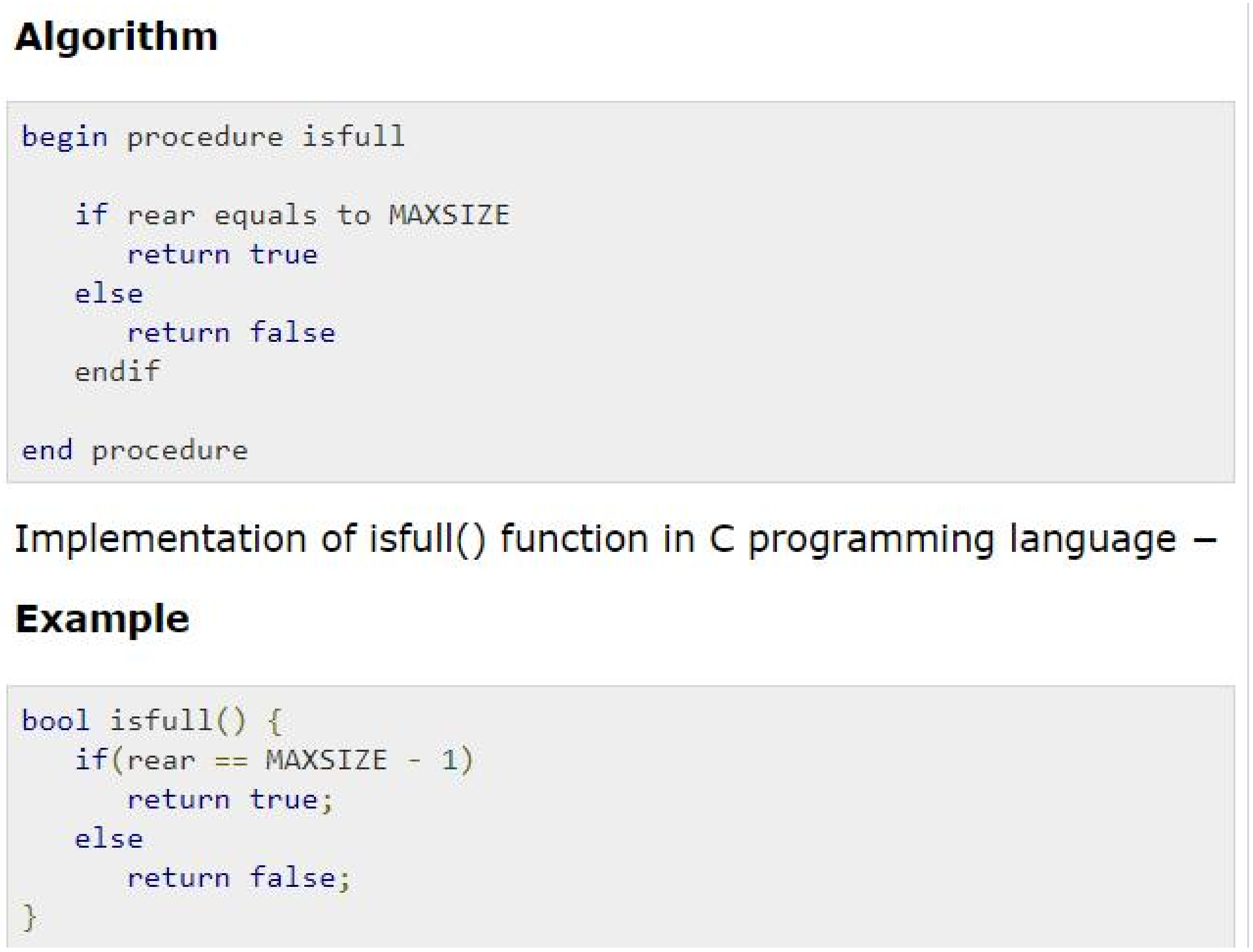
Implementation of peek() function in C programming language −

Example

```c
int peek() {
   return queue[front];
}
```

isfull()

- Using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full.
- In case we maintain the queue in a circular linked-list, the algorithm will differ.



Algorithm

```
begin procedure isfull

   if rear equals to MAXSIZE
      return true
   else
      return false
   endif

end procedure
```

Implementation of isfull() function in C programming language –

Example

```
bool isfull() {
   if(rear == MAXSIZE - 1)
      return true;
   else
      return false;
}
```

isempty()

Algorithm

```
begin procedure isempty

   if front is less than MIN  OR front is greater than rear
      return true
   else
      return false
   endif

end procedure
```

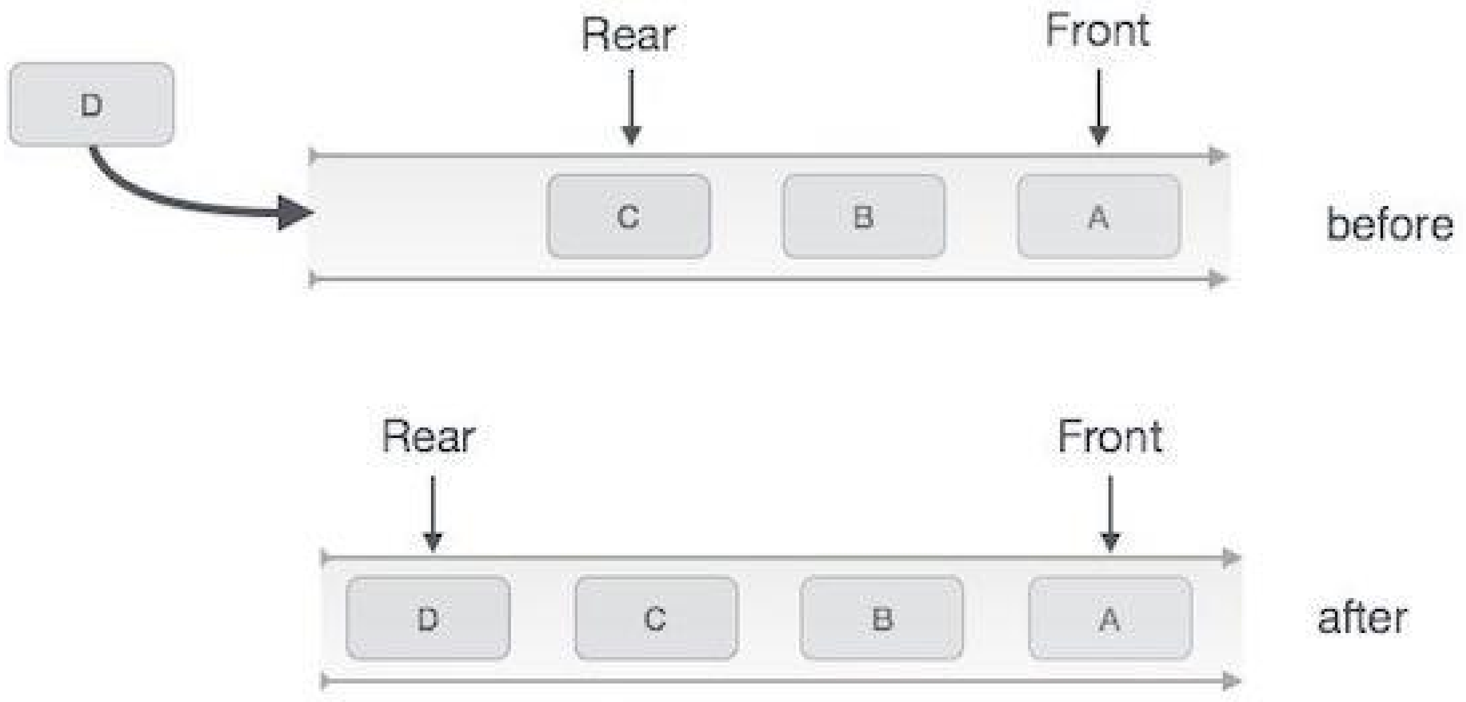
```
bool isempty() {
   if(front < 0 || front > rear)
      return true;
   else
      return false;
}
```

If the value of **front** is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.

Enqueue Operation

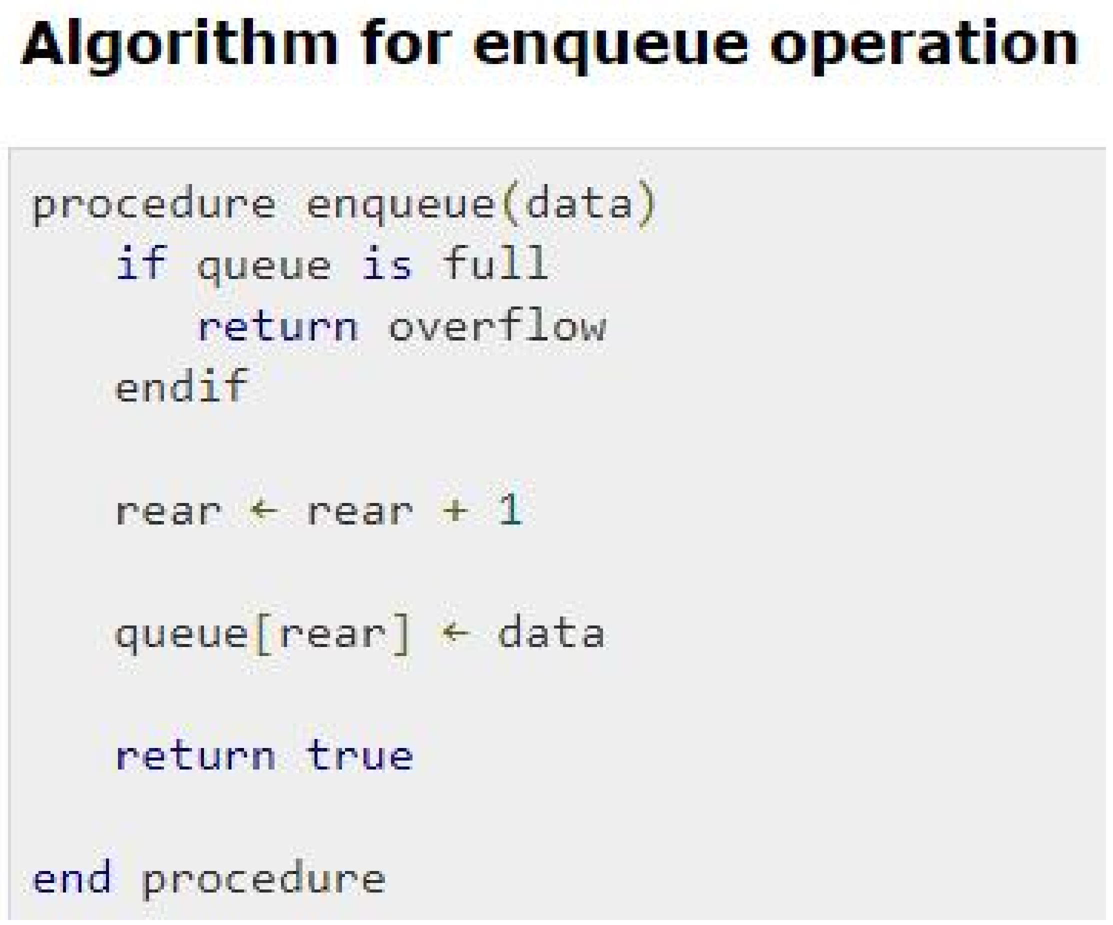
- Queues maintain two data pointers, **front** and **rear**. Therefore, its operations are comparatively difficult to implement than that of stacks.
- The following steps should be taken to enqueue (insert) data into a queue −
- **Step 1** − Check if the queue is full.
- **Step 2** − If the queue is full, produce overflow error and exit.
- **Step 3** − If the queue is not full, increment **rear** pointer to point the next empty space.
- **Step 4** − Add data element to the queue location, where the rear is pointing.
- **Step 5** − return success.

Enqueue Operation



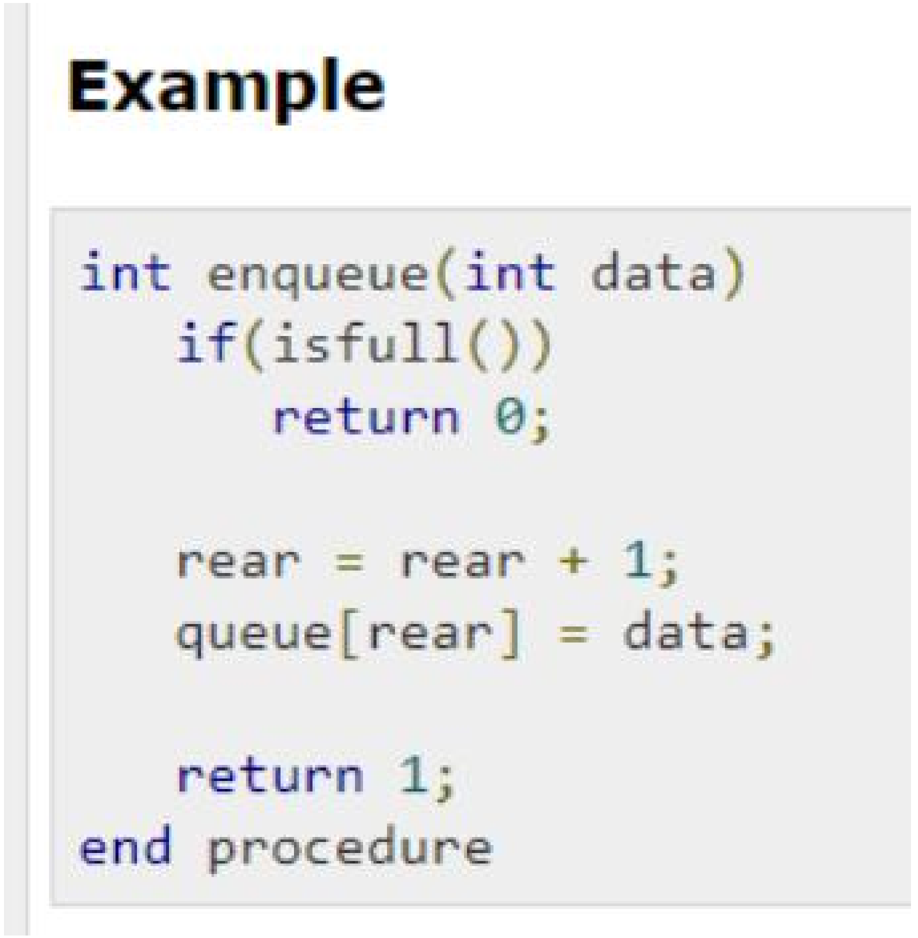
Queue Enqueue

Enqueue Operation



Algorithm for enqueue operation

```
procedure enqueue(data)
   if queue is full
      return overflow
   endif

   rear ← rear + 1

   queue[rear] ← data

   return true

end procedure
```



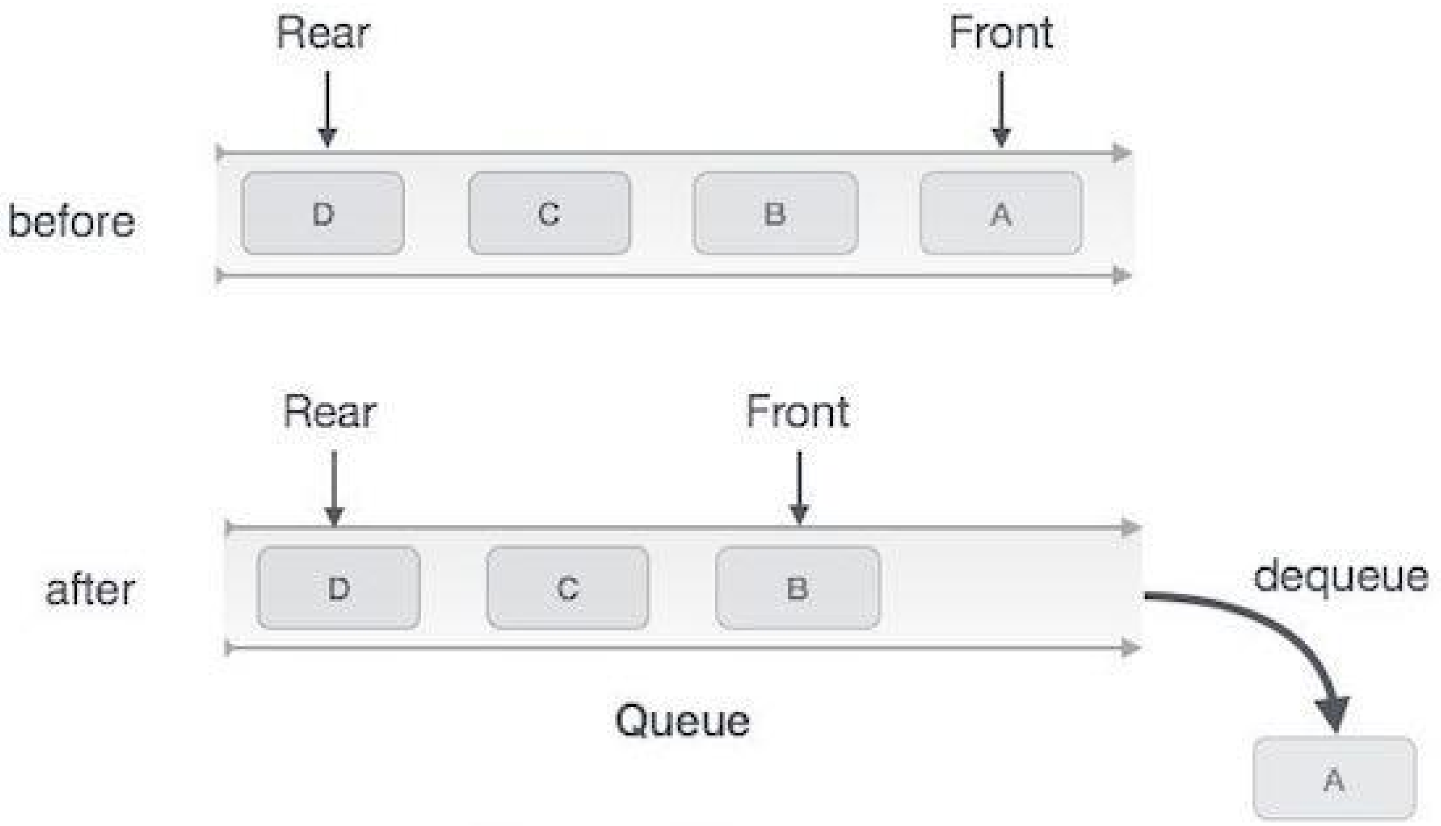
Example

```
int enqueue(int data)
   if(isfull())
      return 0;

   rear = rear + 1;
   queue[rear] = data;

   return 1;
end procedure
```

Dequeue Operation

- Accessing data from the queue is a process of two tasks − access the data where **front** is pointing and remove the data after access.
- The following steps are taken to perform **dequeue** operation −
- **Step 1** − Check if the queue is empty.
- **Step 2** − If the queue is empty, produce underflow error and exit.
- **Step 3** − If the queue is not empty, access the data where **front** is pointing.
- **Step 4** − Increment **front** pointer to point to the next available data element.
- **Step 5** − Return success.

Dequeue Operation



Queue Dequeue

Dequeue Operation

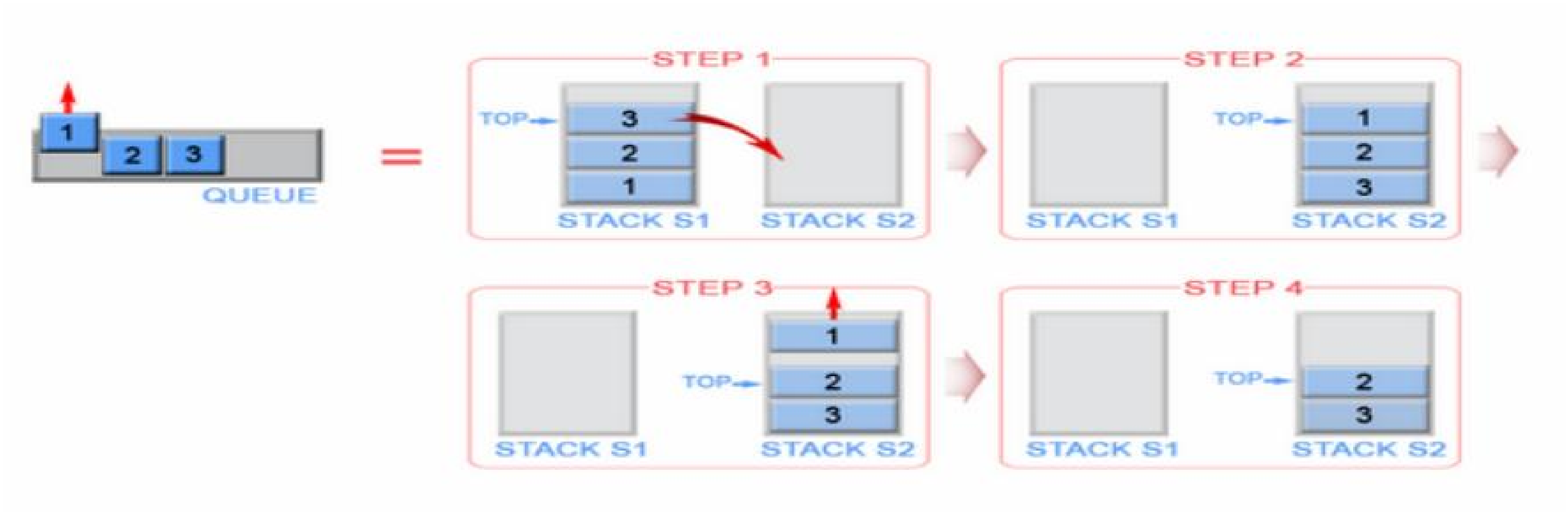
Algorithm for dequeue operation

```
procedure dequeue
   if queue is empty
      return underflow
   end if

   data = queue[front]
   front ← front + 1

   return true
end procedure
```

Example

```
int dequeue() {

   if(isempty())
      return 0;

   int data = queue[front];
   front = front + 1;

   return data;
}
```

How to implement a queue using two stacks?



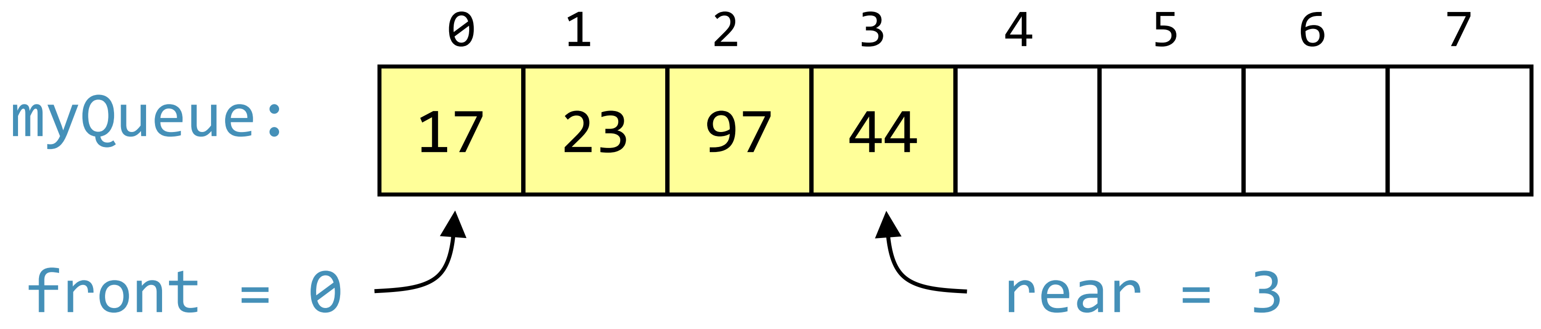
Popping element "1" from the queue

Summary

- Queue
- Operations

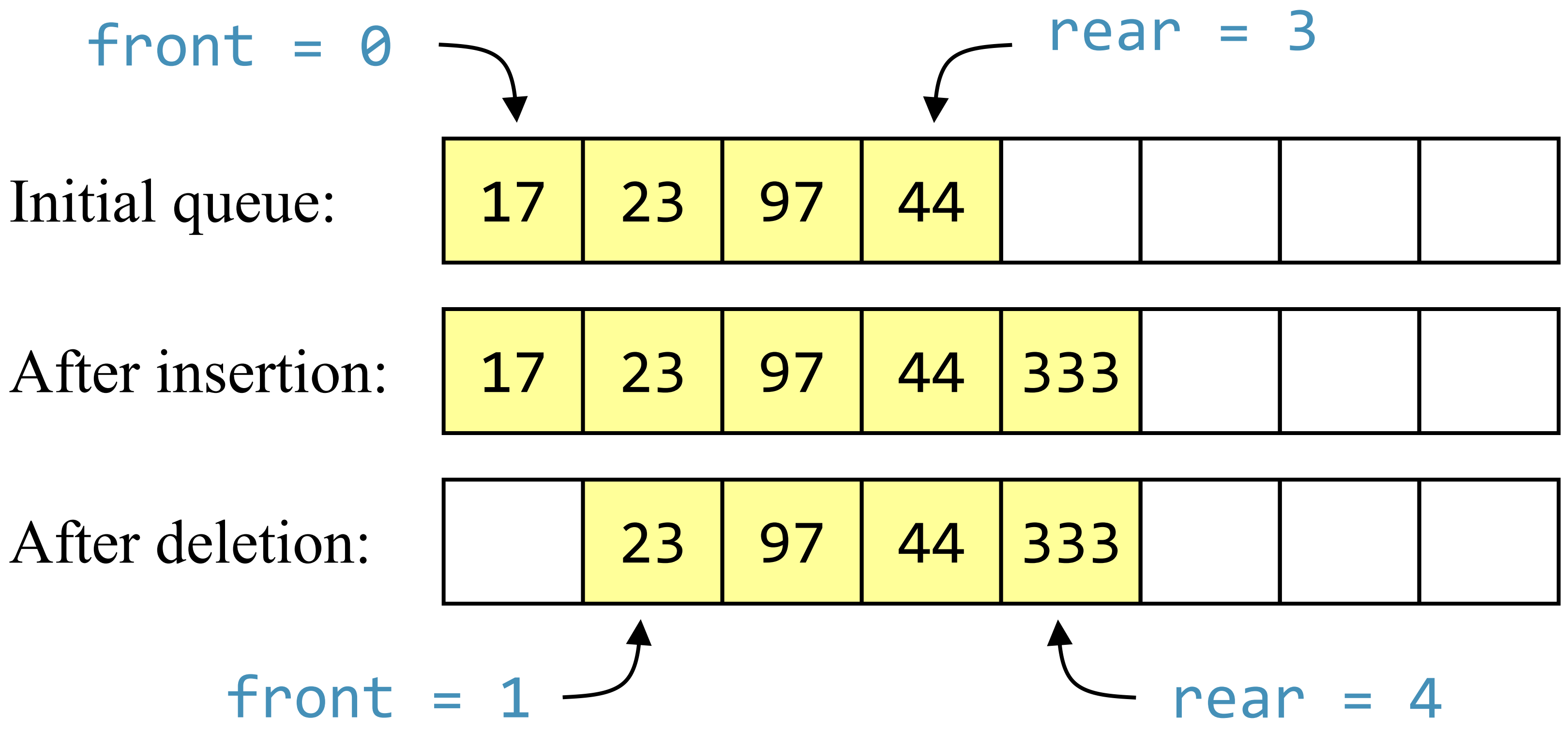
ARRAY IMPLEMENTATION OF QUEUES

- A queue is a first in, first out (FIFO) data structure
- This is accomplished by inserting at one end (the rear) and deleting from the other (the front)



- **To insert:** put new element in location 4, and set rear to 4
- **To delete:** take element from location 0, and set front to 1

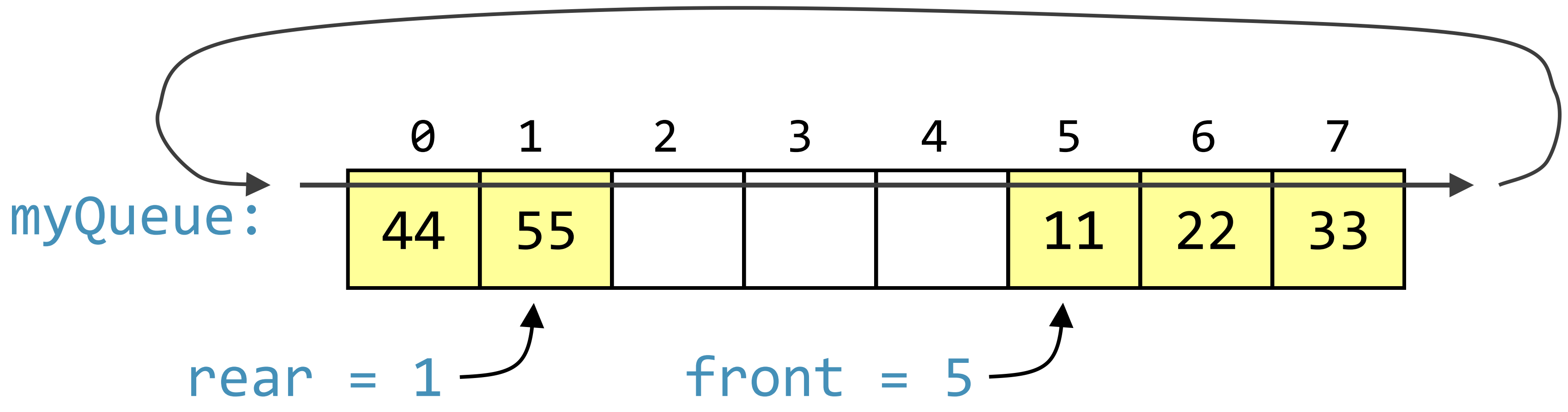
ARRAY IMPLEMENTATION OF QUEUES



- Notice how the array contents "crawl" to the right as elements are inserted and deleted
- This will be a problem after a while!

CIRCULAR ARRAYS

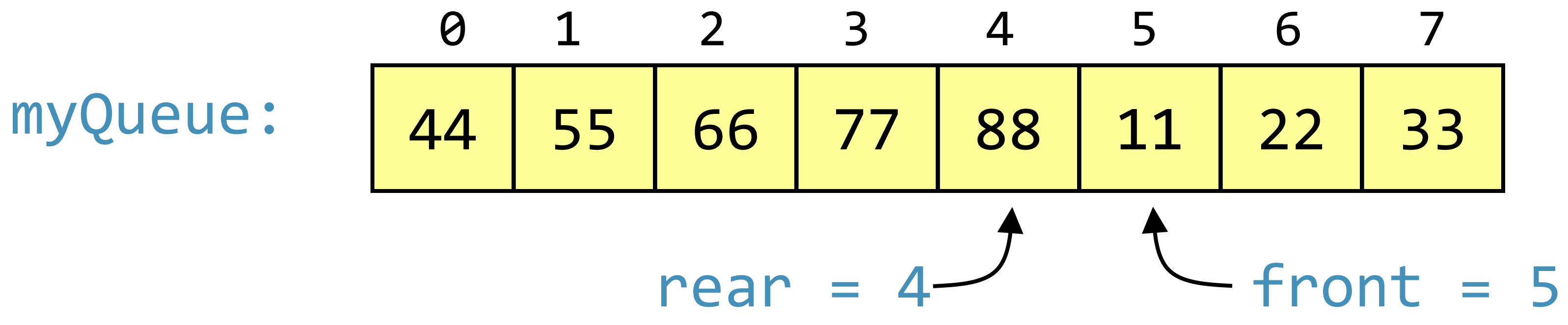
- We can treat the array holding the queue elements as circular (joined at the ends)



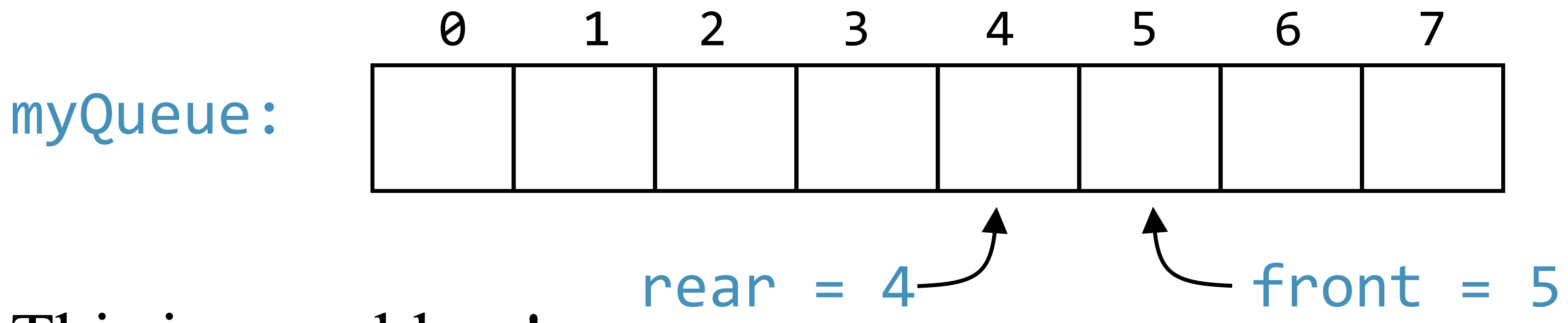
- Elements were added to this queue in the order 11, 22, 33, 44, 55, and will be removed in the same order
- Use: `front = (front + 1) % myQueue.length;` and: `rear = (rear + 1) % myQueue.length;`

FULL AND EMPTY QUEUES

- If the queue were to become completely full, it would look like this:



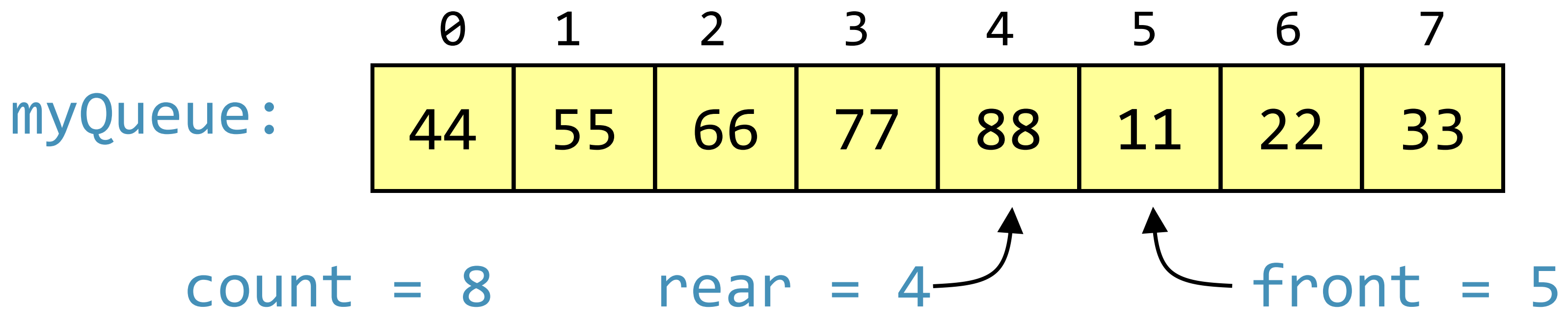
- If we were then to remove all eight elements, making the queue completely empty, it would look like this:



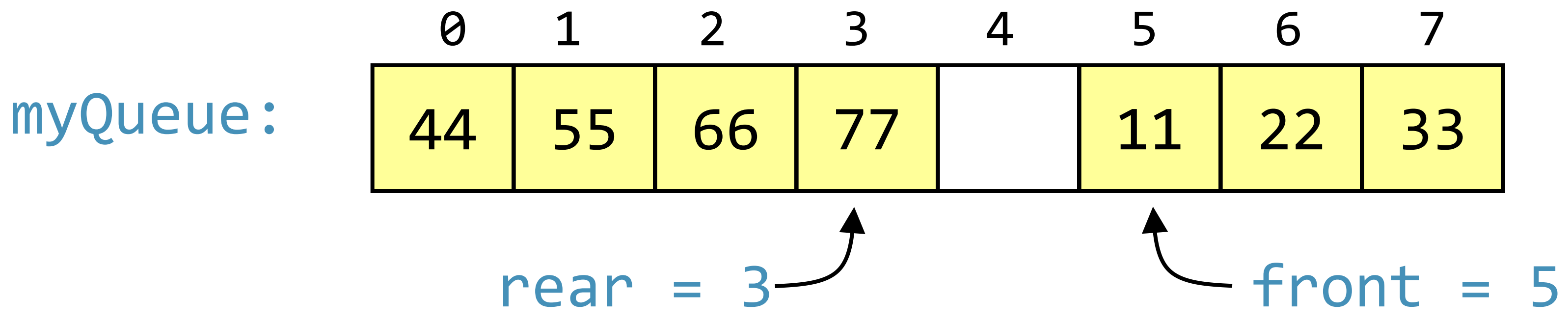
This is a problem!

FULL AND EMPTY QUEUES: SOLUTIONS

- **Solution #1:** Keep an additional variable



- **Solution #2:** (Slightly more efficient) Keep a gap between elements: consider the queue full when it has `n-1` elements



0 1 2 3 4 5 6 7
myQueue:
44 55 66 77 11 22 33
rear = 3
front = 5