Queues

* a linear structure which follows a particular order in which the operations are performed
* Queues are First In First Out (FIFO)
* Think of the real world example of a grocery store line. The customer who is first in line is the customer who purchases their items and leaves the line first

# Operations

1. enqueue(x) or push(x)→ adds x to queue →O(1)

2. dequeue() or pop() → remove the element at front of queue → O(1)

3. front() or peek() → returns element at front of queue → O(1)

4. rear() → returns element at back of queue → O(1)

5. isEmpty() → checks if queue is empty or not → O(1)

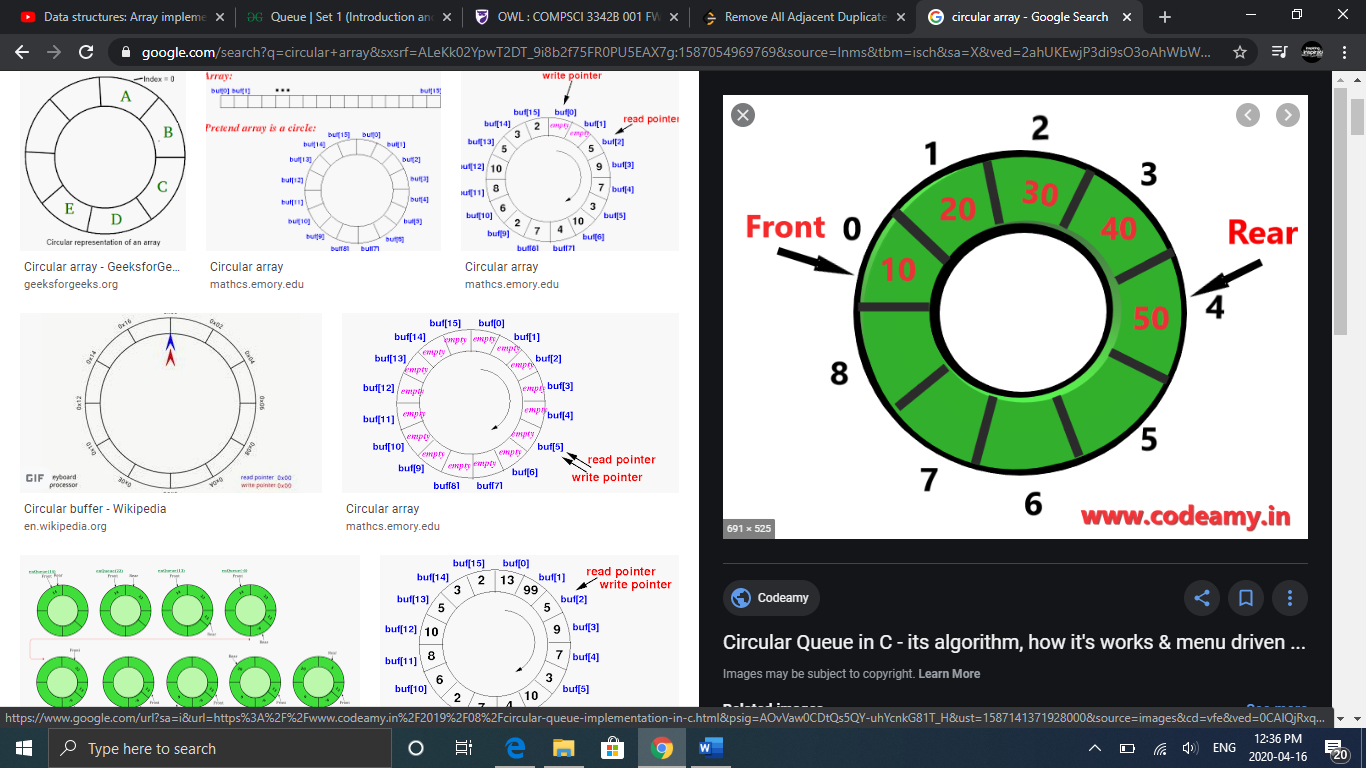
6. isFull() → checks if the queue is full, if queue size is limited → O(1)

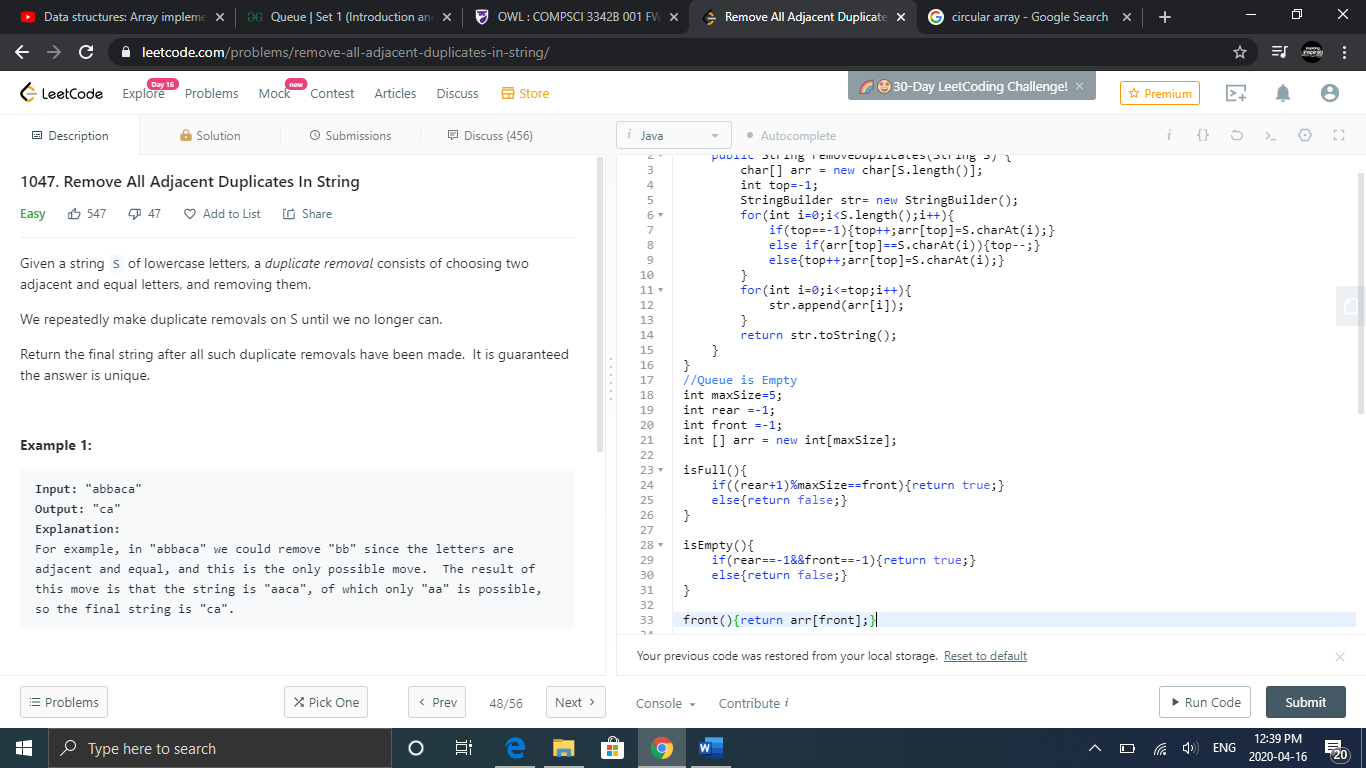
# Applications

* Used in Breadth First Search Algorithm

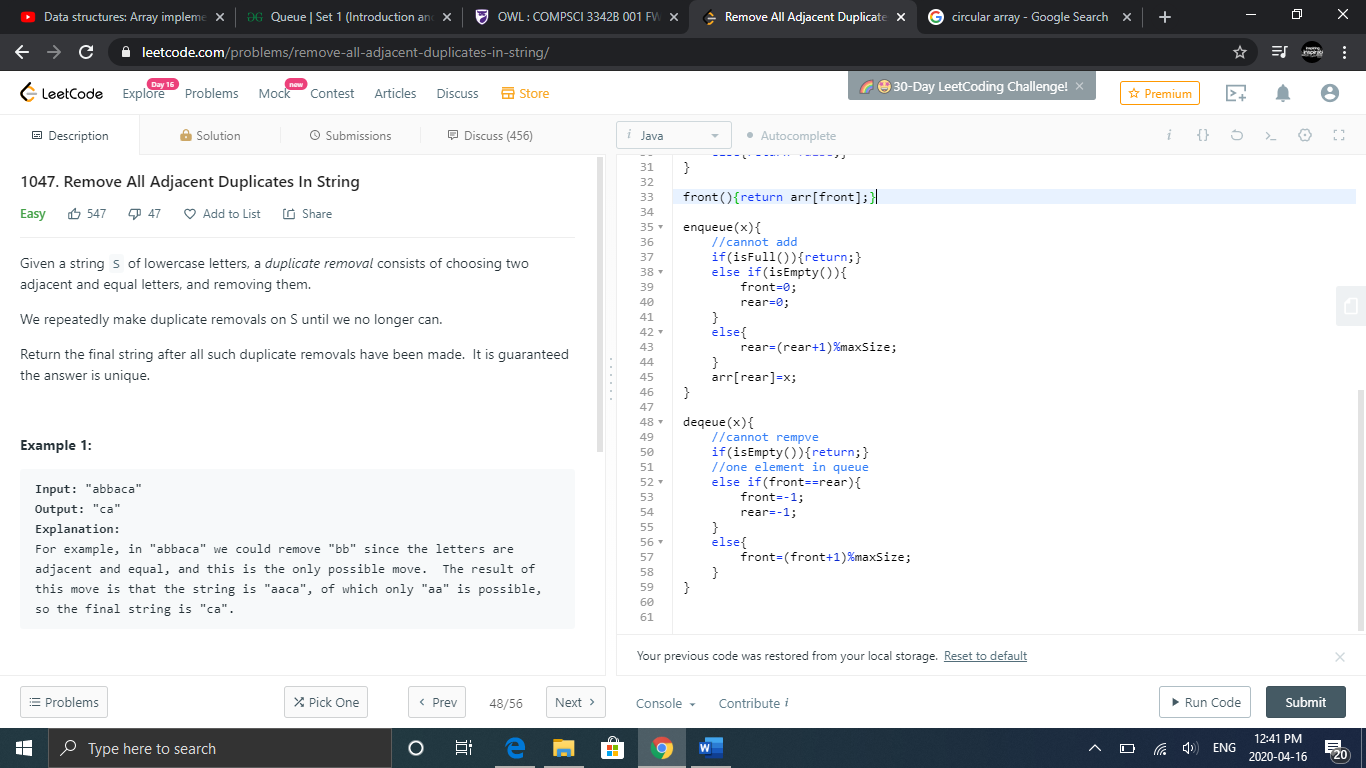
# Array Implementation

* We will use a circular array to implement the queue
* If we use a normal array, every time we dequeue (front+1) we will waste space in the array because front never decreases in the implementation
* In a circular array:
* Current position = i
* Next position = (i+1)%arraySize
* Previous position = (i+arraySize-1)%arraySize
* Visual Representation:

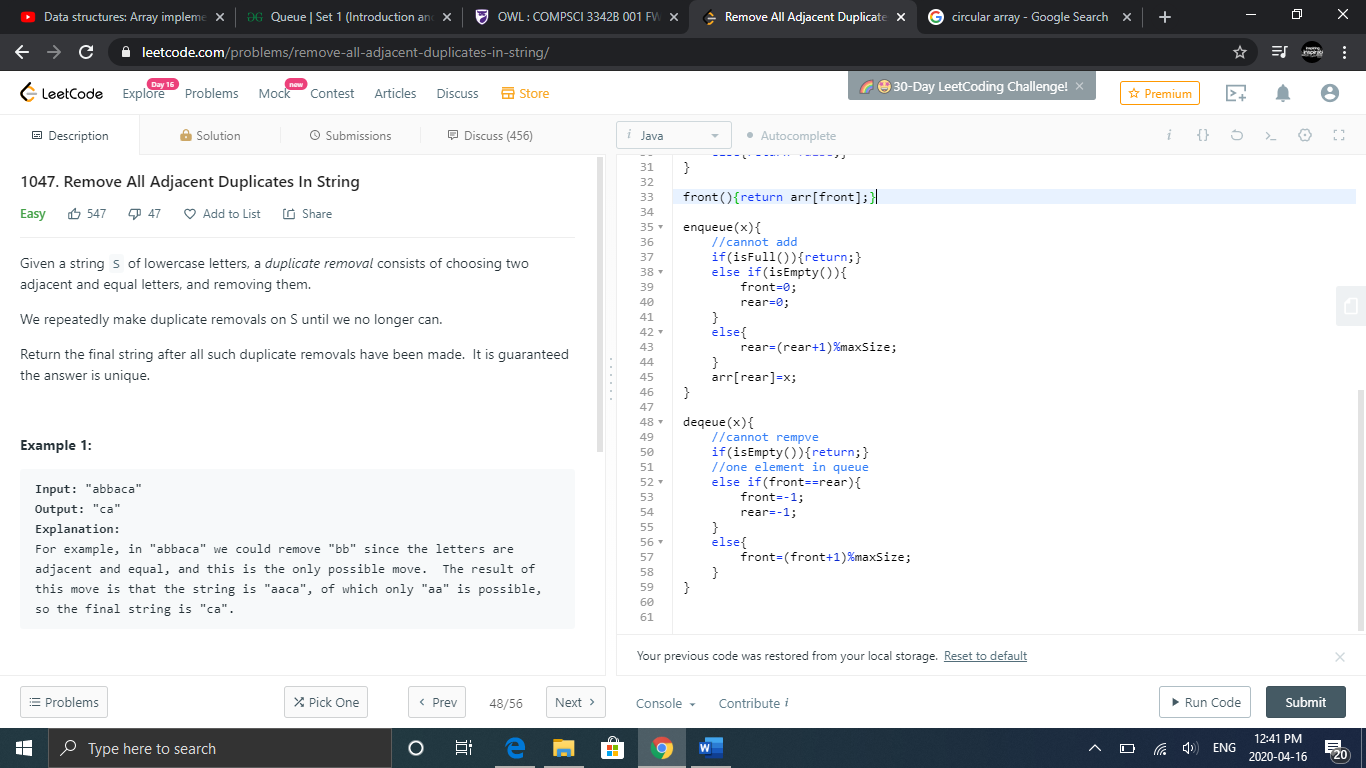




**Enqueue**



**Dequeue**



* A problem with the array implementation is that we may have a lot of unused memory if the array only gets filled with a few elements

# Linked List Implementation

* LinkedList queue = new LinkedList<>(); //linked list in java represents queue
* enqueue(x) → add(x) //adds x to end of the list
* front() → getFirst()
* dequeue() →remove() //removes first element in list
* isEmpty() → if(queue.size()==0){return true;} else{return false;}
* isFull() → dynamic list so there is no max capacity
* You will need two pointers (front and rear)
* Java has the getFirst() and getLast() methods which allows you to avoid making these pointers but if you do not use java you may need to have these pointers
* By having these two pointers, your enqueue and dequeue operations will take O(1) time

