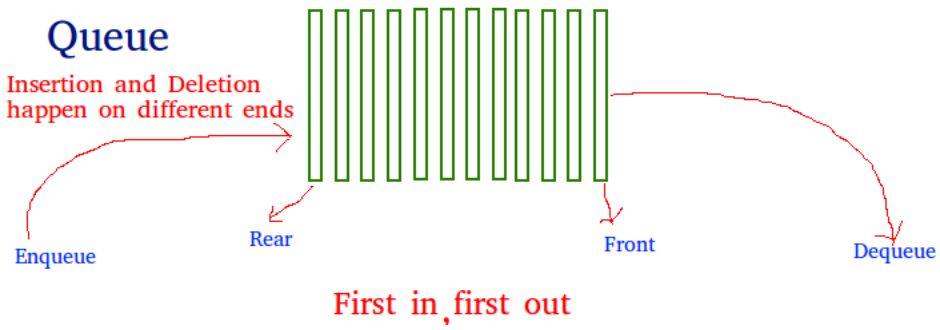
# QUEUE

Queue is a linear data structure with a rear and a front end, similar to a stack. It stores items sequentially in a FIFO (First In First Out) manner. You can think of it as a customer services queue that functions on a first-come-first-serve basis.




**Operations associated with queue are:**

→**Enqueue:** Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition – Time Complexity: O(1)

→**Dequeue***:* Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition – Time Complexity: O(1)

→**Front:** Get the front item from queue – Time Complexity: O(1)

→Rear: Get the last item from queue – Time Complexity: O(1)

**Queue Methods in Python:**

There are numerous methods available in Python to perform operations on the queue. Some of the standard methods are:

→put(item): Inserts an element to the queue
→get(): Gets an element from the queue
→empty(): Checks and returns true if the queue is empty
→qsize: Returns queue's length
→full(): Checks and returns true if the queue is full
→maxsize(): Maximum elements allowed in a queue

**Implementing a Queue in Python with a List:**

Python list is used as a way of implementing queues. The list's append() and pop() methods can insert and delete elements from the queue. However, while using this method, shift all the other elements of the list by one to maintain the FIFO manner. This results in requiring O(n) time complexity. The example below demonstrates a Python queue using a list.

```
# Initialize a queue

queue_exm = []

# Adding elements to the queue

queue_exm.append('x')
queue_exm.append('y')
queue_exm.append('z')
print("Queue before any operations")
print(queue_exm)

# Removing elements from the queue

print("\nDequeuing items")
print(queue_exm.pop(0))
print(queue_exm.pop(0))
print(queue_exm.pop(0))
print("\nQueue after deque operations")
print(queue_exm)
```

**Output:**

Queue before any operations
['x', 'y', 'z']

Dequeuing items
x
y
z

Queue after deque operations
[]

**Implementing a Queue in Python with collections.deque:**

Collections.deque provides the same O(1) time complexity as queues. Hence, it implements a queue, and performs append() & pop() functions quicker than lists. For performing enqueuing and dequeuing using collections.deque, append() and popleft() functions are used.

```
from collections import deque

queue_exm = deque()
queue_exm.append('x')
queue_exm.append('y')
queue_exm.append('z')
print("Queue before operations")
print(queue_exm)
```

```
# Dequeuing elements

print("\nDequeuing elements")
print(queue_exm.popleft())
print(queue_exm.popleft())
print(queue_exm.popleft())
print("\nQueue after operations")
print(queue_exm)
```

**Output:**

Queue before any operations
['x', 'y', 'z']

Dequeuing items
x
y
z

Queue after deque operations
[]

**Adding Elements to a Queue in Python:**

Adding elements to a Python queue from the rear end. The process of adding elements is known as enqueuing. Depicted below is an example to understand it. In this example, we will create a Queue class and use the insert method to implement a FIFO queue.

```
class Queue:

  def __init__(self):
     self.queue = list()

  def element_add_exm(self,data):
# Using the insert method

     if data not in self.queue:
        self.queue.insert(0,data)
        return True
     return False

  def leng(self):
     return len(self.queue)

Queue_add = Queue()
Queue_add.element_add_exm("Mercedes Benz")
Queue_add.element_add_exm("BMW")
```

```
Queue_add.element_add_exm("Maserati")
Queue_add.element_add_exm("Ferrari")
Queue_add.element_add_exm("Lamborghini")
print("Queue's Length: ",Queue_add.leng())
```

**Output:** Queue's Length: 5

**Removing Elements From a Queue in Python:**

Removing an element from a queue, and that process is called dequeuing. Use the built-in pop() function.

```
class Queue:

  def __init__(self):

      self.queue = list()

  def element_add_exm(self,data):

# Using the insert method

      if data not in self.queue:
          self.queue.insert(0,data)
          return True
      return False

# Removing elements

  def element_remove_exm(self):
      if len(self.queue)>0:
          return self.queue.pop()
      return ("Empty Queue")

queu = Queue()
queu.element_add_exm("A")
queu.element_add_exm("B")
queu.element_add_exm("C")
queu.element_add_exm("D")
print(queu) #To print the location of the Queue
print(queu.element_remove_exm())
print(queu.element_remove_exm())
```

**Output:**

<__main__.Queue object at 0x10faca420>
A
B

**Sorting a Queue:**

```
import queue

queu = queue.Queue()

queu.put(5)
queu.put(24)
queu.put(16)
queu.put(33)
queu.put(6)

# Using bubble sort algorithm for sorting

i = queu.qsize()

for x in range(i):     # Removing elements
    n = queu.get()
    for j in range(i-1):    # Removing elements
        y = queu.get()
        if n > y:    # putting smaller elements at beginning
            queu.put(y)
        else:
            queu.put(n)
            n = y
    queu.put(n)
while (queu.empty() == False):
    print(queu.queue[0], end = " ")
    queu.get()
```

**Output:** 5 6 16 24 33

## LeetCode Problems

### Problem: Implement Queue using Stacks

Implement a first in first out (FIFO) queue using only two stacks. The implemented queue should support all the functions of a normal queue (push, peek, pop, and empty).

**Input**
["MyQueue", "push", "push", "peek", "pop", "empty"]
[[], [1], [2], [], [], []]
**Output**
[null, null, null, 1, 1, false]

**Code:**

```
class MyQueue:

    def __init__(self):
        self.queue = []
        self.stack = []

    def push(self, x: int) -> None:
        self.stack.append(x)

    def pop(self) -> int:
        if not self.queue:
            while self.stack:
                self.queue.append(self.stack.pop())
        return self.queue.pop()

    def peek(self) -> int:
        return self.queue[-1] if self.queue else self.stack[0]

    def empty(self) -> bool:
        return not self.queue and not self.stack
```

**Explanation:**

→ MyQueue: Initialize the queue.

stack = []
queue = []

→ push(1): Push 1 onto the queue.

stack = [1]
queue = []

→ push(2): Push 2 onto the queue.

stack = [1, 2]
queue = []

→ peek(): Peek at the front of the queue.

stack = [1, 2]
queue = []

→ pop(): Pop the front element from the queue.

stack = []
queue = [2]

→ empty(): Check if the queue is empty.

stack = []
queue = [2]

**Output:** [null, null, null, 1, 1, false]

-→ Time complexity: O(1)
-→Space complexity: O(n)

**Problem: Time Needed to Buy Tickets**

There are n people in a line queuing to buy tickets, where the 0th person is at the front of the line and the (n - 1)th person is at the back of the line.

You are given a 0-indexed integer array tickets of length n where the number of tickets that the ith person would like to buy is tickets[i].

Each person takes exactly 1 second to buy a ticket. A person can only buy 1 ticket at a time and has to go back to the end of the line (which happens instantaneously) in order to buy more tickets. If a person does not have any tickets left to buy, the person will leave the line.

Return the time taken for the person at position k (0-indexed) to finish buying tickets.

Input: tickets = [2,3,2], k = 2
Output: 6

**Code:**

```
class Solution:
    def timeRequiredToBuy(self, tickets: List[int], k: int) -> int:
        i = 0
        time = 0
        while True:
            if tickets[i] > 0:
                time += 1
                tickets[i] -= 1
            if tickets[k] == 0:
                return time
            i = (i + 1) % len(tickets)
```

**Explanation:**

tickets = [2, 3, 2], k = 2, time = 0

→ i = 0: tickets = [1, 3, 2], time = 1

→ i = 1: tickets = [1, 2, 2], time = 2

→ i = 2: tickets = [1, 2, 1], time = 3

→ i = 0: tickets = [0, 2, 1], time = 4

→ i = 1: tickets = [0, 1, 1], time = 5

→ i = 2: tickets = [0, 1, 0], time = 6

-→ Time Complexity: O(n)
-→ Space Complexity: O(1)

**Problem: Number of Recent Calls**

You have a RecentCounter class which counts the number of recent requests within a certain time frame.

Implement the RecentCounter class:

RecentCounter() Initializes the counter with zero recent requests.
int ping(int t) Adds a new request at time t, where t represents some time in milliseconds, and returns the number of requests that has happened in the past 3000 milliseconds (including the new request). Specifically, return the number of requests that have happened in the inclusive range [t - 3000, t].
It is guaranteed that every call to ping uses a strictly larger value of t than the previous call.

Example 1:

Input
["RecentCounter", "ping", "ping", "ping", "ping"]
[[], [1], [100], [3001], [3002]]
Output
[null, 1, 2, 3, 3]

**Code:**

```
class RecentCounter:
```

```
def __init__(self):
    self.recentCounter = 0
    self.array = []

def ping(self, t: int) -> int:
    self.recentCounter = 0
    self.array.append(t)
    x = t - 3000
    for y in self.array:
        if y >= x and y <= t:
            self.recentCounter+=1
    return self.recentCounter
```

**Explanation:**

→ RecentCounter() self.array = [] and self.recentCounter = 0.
Output: [null]

→ ping(1): t = 1, array = [1].
1 is within the range [1-3000, 1]
Output: [null, 1]

→ ping(100): t = 100, array = [1, 100].
1 and 100 are within the range [100-3000, 100]
Output: [null, 1, 2]

→ ping(3001): t = 3001, array = [1, 100, 3001].
all three values are within the range [3001-3000, 3001]
Output: [null, 1, 2, 3]

→ ping(3002): t = 3002, array = [1, 100, 3001, 3002].
three values (100, 3001, 3002) are within the range [3002-3000, 3002]

**Output:** [null, 1, 2, 3, 3]

-→ Time Complexity: O(n)
-→ Space Complexity: O(n)

**Problem: First Unique Character in a String**

Given a string s, find the first non-repeating character in it and return its index. If it does not exist, return -1.

Input: s = "leetcode"
Output: 0

## Code:

```
class Solution:
    def firstUniqChar(self, s: str) -> int:
        ll=[]
        for i in range(len(s)):
            if s[i] not in ll:
                if s.count(s[i])==1:
                    return i
                ll.append(s[i])
        return -1
```

## Explanation:

→ ll = []

→ i = 0: 'l'
'l' is not in ll -> Check s.count('l')
s.count('l') is 1

Return i = 0

**Output:** Index = 0

-→ Time Complexity: $O(n^2)$
-→ Space Complexity: O(n)

## Challenges faced to solve Queue leet code:

### → Understanding the Problem Requirements

- **Challenge**: Sometimes, the problem statement might be ambiguous or have edge cases that are not immediately obvious. For instance, understanding how the queue should behave in specific scenarios, such as when it's empty or full, can be tricky.
- **Solution**: Carefully read the problem statement and review any provided examples. It might also help to write out the expected behavior in my own words before starting to code.

### → Handling Edge Cases

- **Challenge**: Handling edge cases, such as an empty queue, a queue with only one element, or operations that might cause overflow, can be challenging. For example, when implementing a circular queue, ensuring that correctly handle the wrap-around can be difficult.
- **Solution**: Consider all possible edge cases before starting the implementation. Write tests specifically for these cases to ensure code handles them correctly.

**→ Implementing Circular Queue**

- **Challenge**: When working with circular queues, managing the pointers (front and rear) can be confusing. Ensuring that they correctly wrap around and identifying when the queue is full versus empty are common challenges.
- **Solution**: Use clear and consistent variable names, and carefully plan the logic for updating the pointers. Drawing diagrams to visualize how the pointers move might also help.

**→ Time Complexity Optimization**

- **Challenge**: Some problems may require to optimize the time complexity of queue operations. For example, ensuring that operations like enqueue, dequeue, or peek occur in constant time (O(1)) might require careful thought and implementation.
- **Solution**: Analyze the problem to determine the most efficient data structures or algorithms that will allow to meet the time complexity requirements. For instance, using a deque or implementing an optimized queue using two stacks.

**→ Managing Multiple Queues or Mixed Data Structures**

- **Challenge**: Some medium-level problems might involve using multiple queues simultaneously or combining queues with other data structures, such as stacks or priority queues. Managing these different structures can be complex.
- **Solution**: Break the problem down into smaller parts and tackle each part individually. Ensure that have a good understanding of how each data structure behaves before combining them in solution.

**→ Dealing with FIFO (First In, First Out) Nature**

- **Challenge**: Understanding and leveraging the FIFO property of queues is essential for solving queue problems. However, it might not always be immediately clear how to use this property to solve a specific problem.
- **Solution**: Practice identifying when a problem can be solved using a queue by recognizing patterns where the order of elements is important. Drawing out examples and simulating the queue operations can clarify how FIFO behavior can be applied.

**→ Managing Memory Usage**

- **Challenge**: Some problems may require to efficiently manage memory, especially when dealing with large inputs or when the queue size can grow dynamically.
- **Solution**: Consider using a fixed-size queue or implementing logic to avoid unnecessary memory usage. In languages like Python, be aware of how memory is managed and optimize data structures accordingly.