**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.

A diagram of a bar

Description automatically generated

**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(n2)

-🡪 Space Complexity: O(n)