Queues

While a stack followed a LIFO pattern, a queue follows **FIFO** (first in first out). The side it adds to is the opposite of the size it removes from.

An example is a line at a fast food restaurant. People leave the line when they finish ordering (from the front) and people enter the line from the back (opposite ends). The first peole that enter the line will be the first ones that leave it (FIFO). In software any system that handles a job on a first come, first serve basis - for example, if multiple people are trying to use a printer at the same time.

Queues are trickier to implement than stacks if you want to maintain good performance. Like a stack, you could just use a dynamic array, but operations on the front of the array (adding or removal) are $O(n)$, where n is the size of the array. If you want to these operations to be $O(1)$, you'll need a more sophisticated implementation.

One way to implement an efficient queue is by using a doubly linked list. Recall that with a doubly linked list, if you have the pointer to a node, you can add or delete at that location in $O(1)$.

A doubly linked list that maintains pointers at both ends with sentinel nodes can implement an efficient queue.

A deque can be used to add or delete from both ends. A pure queue is limited to adding to one end and deleting at the other end.

One common use of a queue is sto implement breadth-first search (BFS). This is by far the most common use but we will look at a few other examples that use it.

Interface guide

Python

```
# Declaration: we will use deque from the collections module
import collections
queue = collections.deque()

# If you want to initialize it with some initial values:
```

```
queue = collections.deque([1, 2, 3])

# Enqueueing/adding elements:
queue.append(4)
queue.append(5)

# Dequeuing/removing elements:
queue.popleft() # 1
queue.popleft() # 2

# Check element at front of queue (next element to be removed)
queue[0] # 3

# Get size
len(queue) # 3
```

JavaScript

```
// JavaScript doesn't have any built-in efficient queue
// We'll just have to use a normal array
let queue = [];

// If you want to initialize it with some initial values:
let queue = [1, 2, 3];

// Enqueueing/adding elements:
queue.push(4);
queue.push(5);

// Dequeuing/removing elements:
queue.shift(); // 1
queue.shift(); // 2

// Check element at front of queue (next element to be removed)
queue[0]; // 3

// Get size
queue.length; // 3
```

Java

```
// Declaration: Java supports multiple implementations, but we will using
// the Queue interface with the LinkedList implementation. Specify the data
type
Queue<Integer> queue = new LinkedList<>();

// If you want to initialize it with some initial values:
Queue<Integer> queue = new LinkedList<>(Arrays.asList(1, 2, 3));

// Enqueueing/adding elements:
queue.offer(4);
queue.offer(5);

// Dequeuing/removing elements:
queue.poll(); // 1
queue.poll(); // 2

// Check if empty
queue.isEmpty(); // false

// Check element at front of queue (next element to be removed)
queue.peek(); // 3

// Get size
queue.size(); // 3
```

C++

```
// Declaration: C++ supports multiple implementations, but we will be using
// std::queue. Specify the data type
queue<int> queue;

// Enqueueing/adding elements:
queue.push(1);
queue.push(2);
queue.push(3);
```

```
// Dequeing/removing elements:
queue.pop();

// Check if empty
queue.empty(); // false

// Check element at front of queue (next element to be removed)
queue.front(); // 2

// Get size
queue.size(); // 2
```

Example: 933 - Number of Recent Calls

Implement the `RecentCounter` class. It should support `ping(int t)`, which records a call at time `t`, and then returns an integer representing the number of calls that have happened in the range `[t - 3000, t]`. Calls to `ping` will have increasing `t`.

Let's say we have a value `x`. Once `t` goes beyond `x + 3000`, every future call to `ping` would have to iterate over `x` if we just use an array even though `x` won't be included. We should get rid of `x` as soon as it is outdated.

Removing from the front with just a dynamic array would cost $O(n)$ for each removal but with an efficient queue instead, those removals become $O(1)$. Let's use a queue to store the numbers, and at each call remove all the outdated elements before returning the count.

```
from collections import deque

class RecentCounter:
    def __init__(self):
        self.queue = deque()

    def ping(self, t: int) -> int:
        while self.queue and self.queue[0] < t - 3000:
            self.queue.popleft()

        self.queue.append(t)
        return len(self.queue)
```

```
# Your RecentCounter object will be instantiated and called as such:
# obj = RecentCounter()
# param_1 = obj.ping(t)
```

SEE PROBLEM 346