_Visualization of the queue data structure _

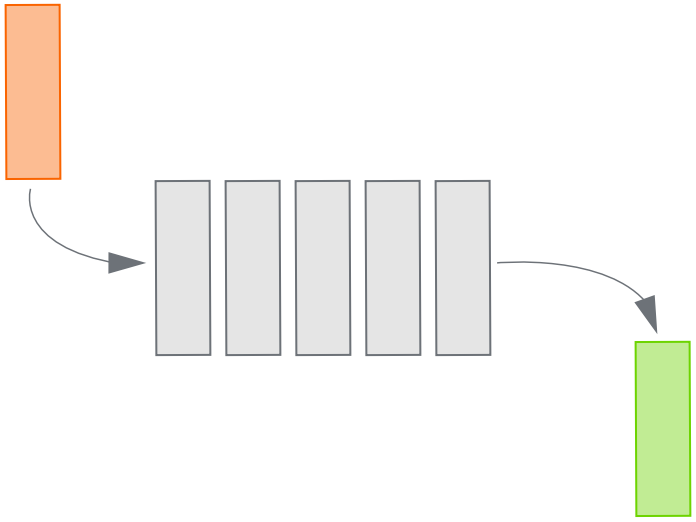

Figure 1: queue.svg

A queue is a sequential data structure and most similar to a stack

A queue is basically a stack inverted and can be visualised as a line of people waiting to be served. The person that is first into the queue is served first. The people who join the queue behind this person will be served after them.

As a result it can be summarised as **'first in, first out'** or FIFO.

Just like a stack it is a sequential data structure without random access. You cannot access all the elements at one instant, you can only access the oldest element. If you wish to access the newest element, you have to move through all the others that are ahead of it first, at which point it becomes the oldest.

It differs from a stack in that a stack only has one point of transaction: the front or 'top' of the stack. With a stack you add and remove from the top or front. With a queue, you have two points of transaction: the front of the queue for removing elements and the back of the queue for adding elements.

We can however add a 'peek' method to see which is the next element in line to come out.

As we are removing the first element added, we use an `array shift` method to remove items from the front of the array.

Removing an element from the queue is called **dequeuing**. Adding an element to the queue is called **enqueuing**. In terms of the tail/head nomenclature, the end of the queue where elements are enqueued is the **tail** and front of the queue, where elements are removed is the **head**.

```
class Queue {
  items = [] // array to store the elements comprising the queue
  enqueue = (element) => this.items.push(element) // add element to back
  dequeue = () => this.items.shift()  // remove element from the front

    // Optional helper methods:
  isEmpty = () => (this.items.length === 0) // return true if the queue is empty
  clear = () => this.items.length = 0 // empty the queue
  size = () => this.items.length // count elements in queue
    peek = () => !this.isEmpty() ? this.items[0] : undefined; // check which element is nex
}
```

Use cases

- A queue sequences data in the order that it was first received. Thus it is most beneficial in scenarios where receipt time is a factor. For example, imagine a service whereby tickets go on sale at a certain time for a limited period. You may want to prioritise those who sent their payment earliest over those who arrived later.
- Serving requests on a single shared resource like a printer or CPU task scheduling.