Stacks and Queues

15 minutes

Stacks and Queues

Stacks and Queues aren't really "data structures" by the strict definition of the term. The more appropriate terminology would be to call them abstract data types (ADTs), meaning that their definitions are more conceptual and related to the rules governing their user-facing behaviors rather than their core implementations.

For the sake of simplicity, we'll refer to them as data structures and ADTs interchangeably throughout the course, but the distinction is an important one to be familiar with as you level up as an engineer.

Now that that's out of the way, Stacks and Queues represent a linear collection of nodes or values. In this way, they are quite similar to the Linked List data structure we discussed in the previous section. In fact, you can even use a modified version of a Linked List to implement each of them. (Hint, hint.)

These two ADTs are similar to each other as well, but each obey their own special rule regarding the order with which Nodes can be added and removed from the structure.

Since we've covered Linked Lists in great length, these two data structures will be quick and easy. Let's break them down individually in the next couple of sections.

What is a Stack?

Stacks are a Last In First Out (LIFO) data structure. The last Node added to a stack is always the first Node to be removed, and as a result, the first Node added is always the last Node removed.

The name Stack actually comes from this characteristic, as it is helpful to visualize the data structure as a vertical stack of items. Personally, I like to think of a Stack as a stack of plates, or a stack of sheets of paper. This seems to make them more approachable, because the analogy relates to something in our everyday lives.

If you can imagine adding items to, or removing items from, a Stack of...literally anything...you'll realize that every (sane) person naturally obeys the LIFO rule.

We add things to the *top* of a stack. We remove things from the *top* of a stack. We never add things to, or remove things from, the *bottom* of the stack. That's just crazy.

Note: We can use JavaScript Arrays to implement a basic stack. `Array#push` adds to the top of the stack and `Array#pop` will remove from the top of the stack. In the exercise that follows, we'll build our own Stack class from scratch (without using any arrays). In an interview setting, your evaluator may be okay with you using an array as a stack.

What is a Queue?

Queues are a First In First Out (FIFO) data structure. The first Node added to the queue is always the first Node to be removed.

The name Queue comes from this characteristic, as it is helpful to visualize this data structure as a horizontal line of items with a beginning and an end. Personally, I like to think of a Queue as the line one waits on for an amusement park, at a grocery store checkout, or to see the teller at a bank.

If you can imagine a queue of humans waiting...again, for literally anything...you'll realize that *most* people (the civil ones) naturally obey the FIFO rule.

People add themselves to the *back* of a queue, wait their turn in line, and make their way toward the *front*. People exit from the *front* of a queue, but only when they have made their way to being first in line.

We never add ourselves to the front of a queue (unless there is no one else in line), otherwise we would be "cutting" the line, and other humans don't seem to appreciate that.

Note: We can use JavaScript Arrays to implement a basic queue. `Array#push` adds to the back (enqueue) and `Array#shift` will remove from the front (dequeue). In the exercise that follows, we'll build our own Queue class from scratch (without using any arrays). In an interview setting, your evaluator may be okay with you using an array as a queue.

Stack and Queue Properties

Stacks and Queues are so similar in composition that we can discuss their properties together. They track the following three properties:

Stack Properties | Queue Properties:

Stack Property	Description	Queue Property	Description
`top`	The first node in the Stack	`front`	The first node in the Queue.
----	Stacks do not have an equivalent	`back`	The last node in the Queue.
`length`	The number of nodes in the Stack; the Stack's length.	`length`	The number of nodes in the Queue; the Queue's length.

Notice that rather than having a `head` and a `tail` like Linked Lists, Stacks have a `top`, and Queues have a `front` and a `back` instead. Stacks don't have the equivalent of a `tail` because you only ever push or pop things off the top of Stacks. These properties are essentially the same; pointers to the end points of the respective List ADT where important actions way take place. The differences in naming conventions are strictly for human comprehension.

Similarly to Linked Lists, the values stored inside a Stack or a Queue are actually contained within Stack Node and Queue Node instances. Stack, Queue, and Singly Linked List Nodes are all identical, but just as a reminder and for the sake of completion, these List Nodes track the following two properties:

Stack & Queue Node Properties:

Property	Description
`value`	The actual value this node represents.
`next`	The next node in the Stack (relative to this node).

Stack Methods

In the exercise that follows, we will implement a Stack data structure along with the following Stack methods:

Type	Name	Description	Returns
Insertion	`push`	Adds a Node to the top of the Stack.	Integer - New size of stack
Deletion	`pop`	Removes a Node from the top of the Stack.	Node removed from top of Stack
Meta	`size`	Returns the current size of the Stack.	Integer

Queue Methods

In the exercise that follows, we will implement a Queue data structure along with the following Queue methods:

Type	Name	Description	Returns
Insertion	`enqueue`	Adds a Node to the front of the Queue.	Integer - New size of Queue
Deletion	`dequeue`	Removes a Node from the front of the Queue.	Node removed from front of Queue
Meta	`size`	Returns the current size of the Queue.	Integer

Time and Space Complexity Analysis

Before we begin our analysis, here is a quick summary of the Time and Space constraints of each Stack Operation.

Data Structure Operation	Time Complexity (Avg)	Time Complexity (Worst)	Space Complexity (Worst)
Access	`Θ(n)`	`O(n)`	`O(n)`

Search	`Θ(n)`	`O(n)`	`O(n)`
Insertion	`Θ(1)`	`O(1)`	`O(n)`
Deletion	`Θ(1)`	`O(1)`	`O(n)`

Before moving forward, see if you can reason to yourself why each operation has the time and space complexity listed above!

Time Complexity - Access and Search:

When the Stack ADT was first conceived, its inventor definitely did not prioritize searching and accessing individual Nodes or values in the list. The same idea applies for the Queue ADT. There are certainly better data structures for speedy search and lookup, and if these operations are a priority for your use case, it would be best to choose something else!

Search and Access are both linear time operations for Stacks and Queues, and that shouldn't be too unclear. Both ADTs are nearly identical to Linked Lists in this way. The only way to find a Node somewhere in the middle of a Stack or a Queue, is to start at the `top` (or the `back`) and traverse downward (or forward) toward the `bottom` (or `front`) one node at a time via each Node's `next` property.

This is a linear time operation, O(n).

Time Complexity - Insertion and Deletion:

For Stacks and Queues, insertion and deletion is what it's all about. If there is one feature a Stack absolutely must have, it's constant time insertion and removal to and from the `top` of the Stack (FIFO). The same applies for Queues, but with insertion occurring at the `back` and removal occurring at the `front` (LIFO).

Think about it. When you add a plate to the top of a stack of plates, do you have to iterate through all of the other plates first to do so? Of course not. You simply add your plate to the top of the stack, and that's that. The concept is the same for removal.

Therefore, Stacks and Queues have constant time Insertion and Deletion via their `push` and `pop` or `enqueue` and `dequeue` methods, O(1).

Space Complexity:

The space complexity of Stacks and Queues is very simple. Whether we are instantiating a new instance of a Stack or Queue to store a set of data, or we are using a Stack or Queue as part of a strategy to solve some problem, Stacks and Queues always store one Node for each value they receive as input.

For this reason, we always consider Stacks and Queues to have a linear space complexity, O(n).

When should we use Stacks and Queues?

At this point, we've done a lot of work understanding the ins and outs of Stacks and Queues, but we still haven't really discussed what we can use them for. The answer is actually...a lot!

For one, Stacks and Queues can be used as intermediate data structures while implementing some of the more complicated data structures and methods we'll see in some of our upcoming sections.

For example, the implementation of the breadth-first Tree traversal algorithm takes advantage of a Queue instance, and the depth-first Graph traversal algorithm exploits the benefits of a Stack instance.

Additionally, Stacks and Queues serve as the essential underlying data structures to a wide variety of applications you use all the time. Just to name a few:

Stacks:

- The Call Stack is a Stack data structure, and is used to manage the order of function invocations in your code.
- Browser History is often implemented using a Stack, with one great example being the browser history object in the very popular React Router module.
- Undo/Redo functionality in just about any application. For example:
    - When you're coding in your text editor, each of the actions you take on your keyboard are recorded by `push`ing that event to a Stack.
    - When you hit cmd + z to undo your most recent action, that event is `pop`ed off the Stack, because the last event that occured should be the first one to be undone (LIFO).
    - When you hit cmd + y to redo your most recent action, that event is `push`ed back onto the Stack.

Queues:

- Printers use a Queue to manage incoming jobs to ensure that documents are printed in the order they are received.
- Chat rooms, online video games, and customer service phone lines use a Queue to ensure that patrons are served in the order they arrive.
    - In the case of a Chat Room, to be admitted to a size-limited room.
    - In the case of an Online Multi-Player Game, players wait in a lobby until there is enough space and it is their turn to be admitted to a game.
    - In the case of a Customer Service Phone Line...you get the point.

In the case of a Customer Service Phone Line...you get the point.

- As a more advanced use case, Queues are often used as components or services in the system design of a service-oriented architecture. A very popular and easy to use example of this is Amazon's Simple Queue Service (SQS), which is a part of their Amazon Web Services (AWS) offering.
    - You would add this service to your system between two other services, one that is sending information for processing, and one that is receiving information to be processed, when the volume of incoming requests is high and the integrity of the order with which those requests are processed must be maintained.

✓ Mark As Complete

Finished with this task? Click **Mark as Complete** to continue to the next page!