Exploration: Queues

Queue Operations

A queue is a data structure that imposes a ***first in, first out*** (or ***FIFO***) ordering on the elements it stores. In other words, unlike a stack, the first element to be removed from a queue is the first one that was placed into it. Before an element can be removed, it must wait for the removal of all other elements that were inserted into the queue before it.

As you might imagine, a queue ADT works much like queues in the real world, such as a line of people waiting to check out at the grocery store, or a line of people waiting to ride a roller coaster at an amusement park. If you've watched any British shows/movies, you may have heard a reference to "get in the queue", which is also a phrase you may have noticed we use during Office Hours. In a real-world line of people, everyone gets on a ride in the order in which they entered the line. The first person in the line is the first who gets to ride, the second person in line is the second who gets to ride, and so forth. Each person in the line needs to wait to ride until all of the people who entered the line before them have had their turn. (This assumes no line cuts, which we saw a lot of at Disneyland Paris - apparently it's a thing there.)

The queue ADT specifies an abstract structure with two ends: a front and a back. Every time a new element is inserted into the queue, it is inserted at the back, and every time an element is removed from the queue, it is removed from the front.

The queue ADT supports two main operations:

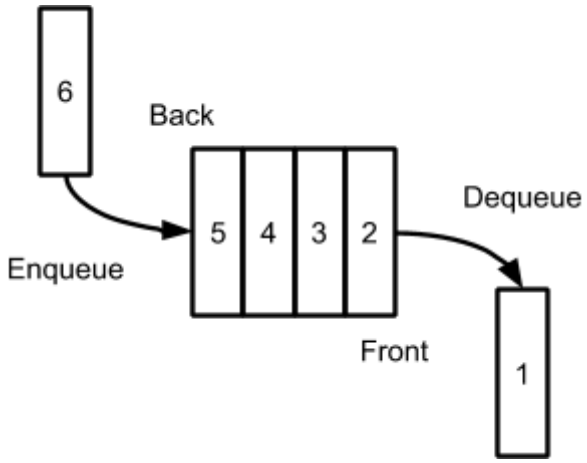
enqueue(x)

Adds `x` to the back of the queue. This item now represents the back of the queue.

dequeue()

Removes the first item in the queue and returns its value. The item behind it is now at the front of the queue.

We can visualize a queue like this:

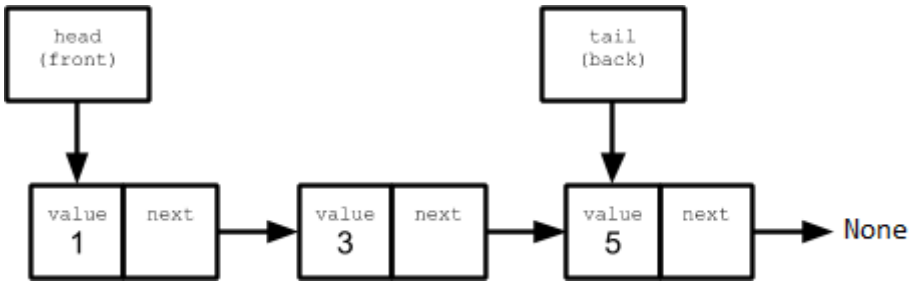


- Unlike the stack, elements are dequeued from a queue in the same order in which they are enqueued. For example, if the values 1, 2, and 3 are enqueued in that order, they will also be dequeued in the same order: 1, 2, 3. For this reason, queues also have many real-world applications, such as scheduling, I/O buffering, etc.

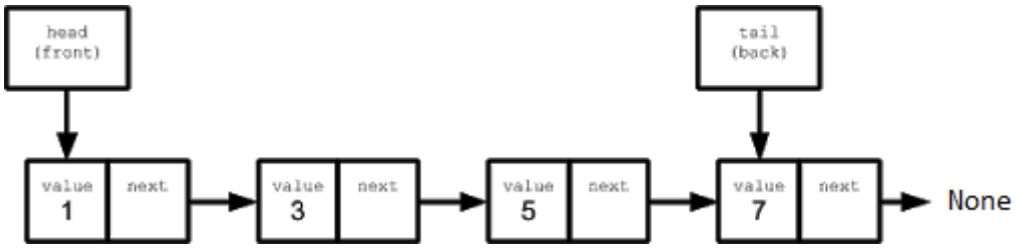
Note also that unlike a stack, where all operations occur at the top, queue operations occur at both ends. This will require us to implement a queue differently than we implement a stack. However, with some modifications, we can still use either a linked list or dynamic array as the underlying storage for a queue.

Implementing a Queue Using a Linked List

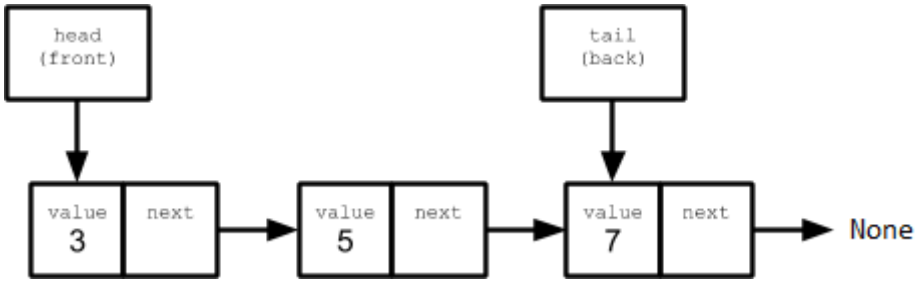
We can implement a queue using a singly linked list. However, since we need to access both ends of a queue (enqueuing to the back and dequeuing from the front), we must keep track of both the head and tail of the list, since those are the two locations where operations occur. Specifically, in a linked list-based implementation of a queue, enqueued values are inserted at the tail while dequeued values are removed from the head. For example, the linked list below represents a queue containing three values: 1 at the front of the queue, followed by 3, with 5 at the back of the queue:



If the user called enqueue(7) on this queue, the value 7 would be inserted at the tail of the underlying list:



If the user then called dequeue() on this queue, the value 1 would be removed from the head of the underlying list:

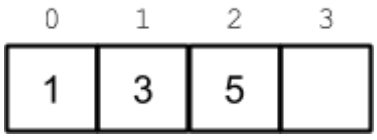


Additional calls to dequeue() would result in the value 3, then 5, and finally 7 being removed from the list.

As with a linked list-based stack, operations on a linked list-based queue (i.e., enqueue() and dequeue()) have *O(1)* runtime complexity (best-case, worst-case, and average), since they only involve the constant-time operations of allocating a single node and updating pointers.

Implementing a Queue Using a Dynamic Array

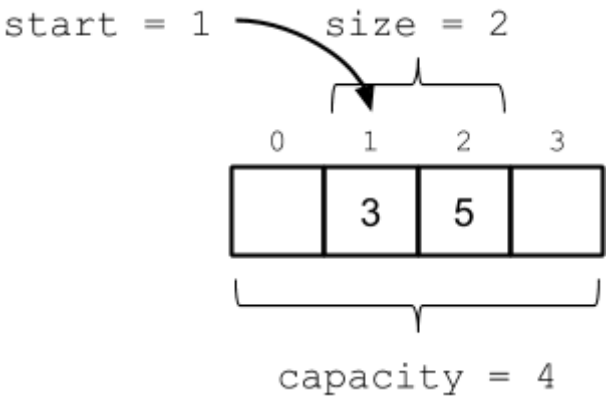
Similarly to using a linked list, implementing a queue using a dynamic array as the underlying data storage requires slight modifications that allow us to easily work with both the front and back of the queue. The front of the queue will correspond to the beginning of the array, and the end of the queue will correspond to the end of the array. For example, the following array represents a queue containing three values: 1 at the front of the queue, followed by 3, and 5 at the back of the queue:



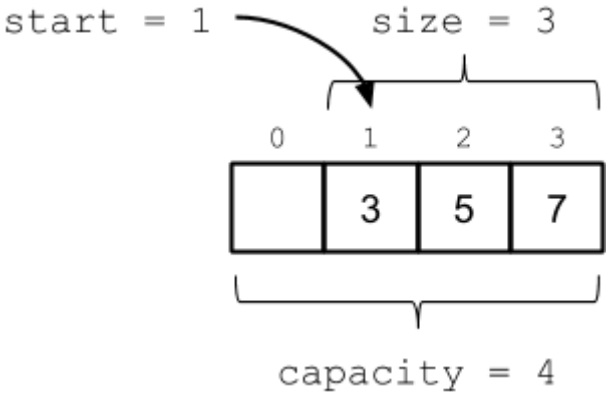
This is straightforward so far. If we want to enqueue a new value here, we can simply insert it at the end of the array. However, what will happen if we want to *dequeue* a value?

If we remove the first value from the underlying array when we dequeue, we will presumably need to move each remaining value forward one index, so that the first element of the queue stays at the front of the array. However, this would mean that each dequeue operation will have *O(n)* runtime complexity, which is undesirable.

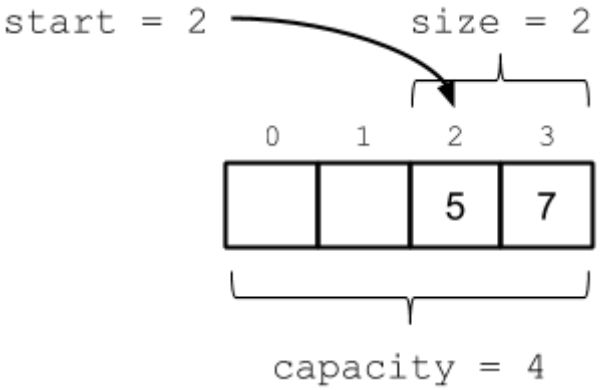
Instead, we will allow the index corresponding to the front of the queue “float” back into the middle of the array. To make this work, we will add an additional field (start) to the dynamic array that keeps track of the first value, e.g.:



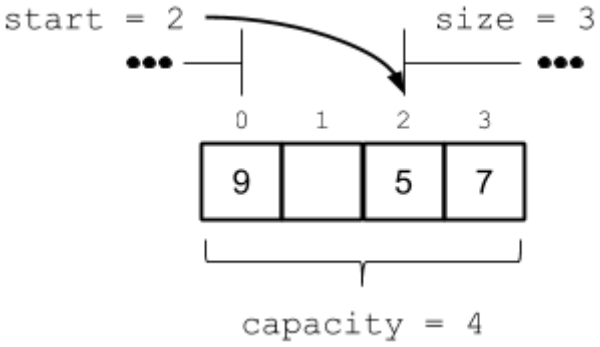
Now, if the user runs the operation enqueue(7), we will still place 7 at the end of the array:



And if the user calls dequeue() again, we would again update the index of the start variable (and decrement the size):



But what do we do if the user now calls enqueue(9)? Do we resize the array, since we’ve reached the end? Doing this could result in inefficient use of memory. Instead, we’ll allow the values to *wrap around* back to the beginning of the array. A call to enqueue(9) would result in an underlying array that looked like this:



An array like this, in which we allow data to wrap around from the back to the front, is known as a ***circular buffer***. A question you may now have, with these modifications, is how do we know which index in the array corresponds to the back of the queue? The answer is straightforward: we maintain a field for the back of the queue, as well as the start and the size.

How do we handle the "wraparound" when we reuse space at the start of the array?

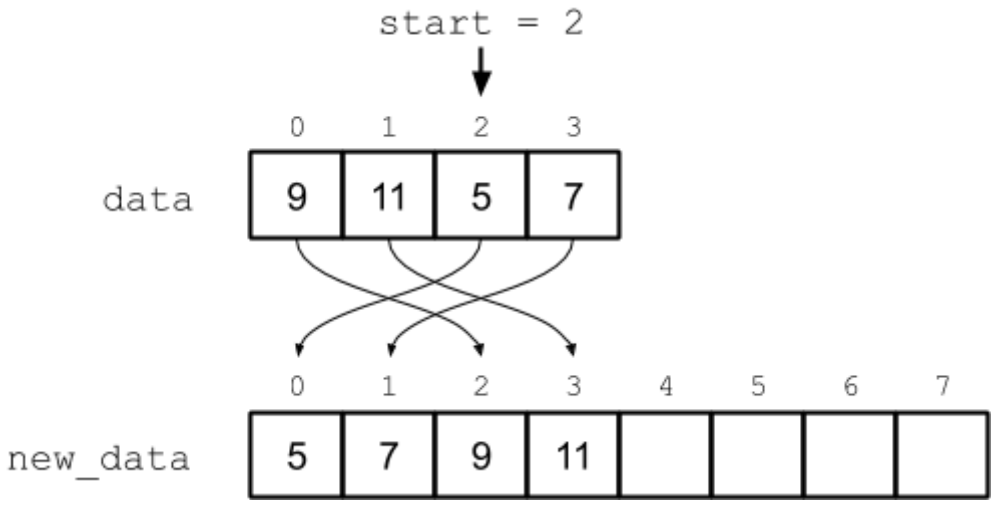
The solution is actually simple and direct: since all queue operations occur at one of two known locations and will only impact their corresponding indices by 1, we check if the incrementing of either the front (after a dequeue) or the back (after an enqueue) exceeds the last valid index in the array. If so, we simply assign it a value of 0. This increment operation can easily be encapsulated into a simple method that increments the index, checks if needs to be set to 0, and returns the result.

Resizing in a dynamic array-based queue implementation

As with the typical dynamic array implementation, we will always resize our queue's dynamic array when its size is equal to its capacity. Note that since the *front* of our queue's underlying array can now contain empty spaces, resizing will not necessarily be needed when we have written data to the end of that array.

When we resize our queue's underlying physical array, we will take the opportunity to ***reindex*** the array so that the start of the queue once again corresponds to the physical index 0. To do this, we loop through the incumbent array beginning at the start index (using the increment logic described above), while copying values to the new array starting at index 0.

Here is a visualization of that process for a full queue with a size and capacity currently at 4, and the start of the queue at index 2:



Since dynamic array insertion and deletion have amortized *O(1)* runtime complexity, even when accounting for resizing, a queue's enqueue() and dequeue() operations will also have amortized *O(1)* average runtime complexity when using a dynamic array as the underlying data storage.

However, since the dynamic array may need to be resized upon any given insertion, a queue's enqueue() operation will have *O(n)* worst-case runtime complexity when built on top of a dynamic array.