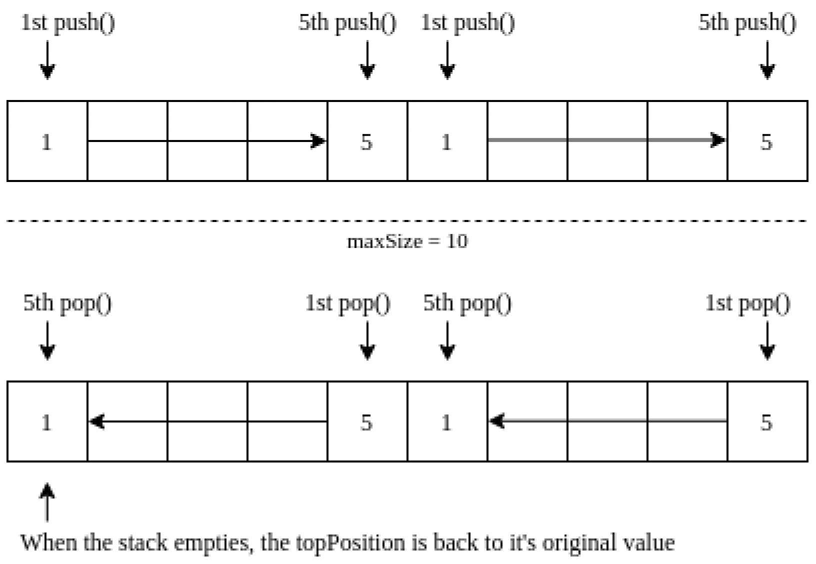
Design & Implementation

Part one required designing and implementing a double stack using an array as the fundamental data structure. This required an array being created of a given maximum size which would be evenly split to create two stacks.

Due to this even splitting, I decided to decrement odd values for the arrays maximum size. Given an odd sized array, there would be one index which could not be used by either stacks or the stacks would not be of even size and so decrementing saves on wasted array space. Incrementing would lead to the double stack being able to hold more elements than the given maximum size.

In the Stack class which takes the underlying array of the double stack, I implemented an isFirstStack flag which determines which half of the array is to be used. I decided to have the first stack initially fill up from index 0 to to half the total array size, and the second stack to fill up from the middle value to the end of the array. This allows me use the same method of increment the topPosition for both the first and second stack to be used as they start from different indexes.



I decided to have a topPosition attribute for the Stack class which pointed to the last added element instead of the next empty position so I could check if the stack was full before trying to add the element . In the same vein, I decided to add a currentSize attribute instead of checking whether the next value is null because this would pose a problem when the stacks are full: If the second stack was empty, the first stack would begin filling up the second stack spaces when full which would mean the stack is violating it's size restriction; When adding to a full second stack, the program would check whether the next value is null of an index which is larger than the size of the array and so would throw a indexOutOfBoundsException.

Having a currentSize also allowed me to easily create the isEmpty method as I could just check whether currentSize was greater than or equal to the maximum size for the stack instead of checking for values themselves, meaning this function has constant time complexity.

Part two required using our double stack implementation to create a queue with a given maximum size. As a queue is first-in-first-out, unlike stacks, I used the hint in the specification to have on of the stacks in the double stack as an input stack, and the other as an output stack. When enqueueing, elements can be added to the input stack. For dequeuing, if the output stack is empty, then all of the elements in the input stack are popped onto the output stack which would effectively reverse the order of the input stack and make the first added element the first one to be dequeued.

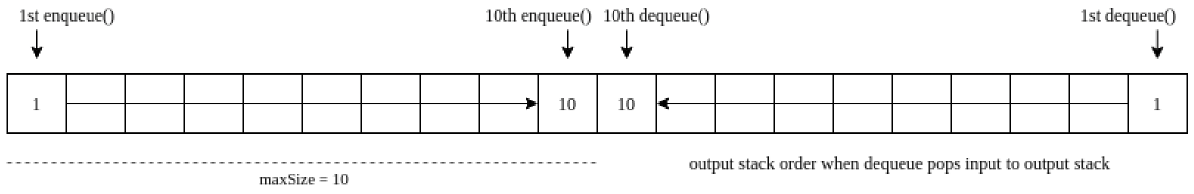


Figure 1: The design for the double stack queue I decided to use where the maximum size is doubled for the creation of the double stack. It shows how the two stacks form the first-in-first-out queue functionality.

I wanted to make sure that the given maximum number of elements could always be added at one time to the queue without having to dequeue elements to do so. So, I created the underlying double stack with an array of double the maximum size of the queue. This meant that the input stack could always hold the maximum number of elements. As double the maximum size would always be an even number, I didn't have to worry about the double stack decrementing the maximum size.

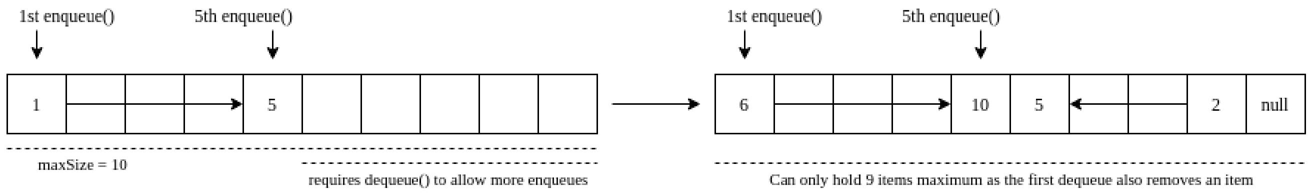

1st enqueue()
5th enqueue()
1
5
maxSize = 10
requires dequeue() to allow more enqueues
1st enqueue()
5th enqueue()
6
10
5
2
null
Can only hold 9 items maximum as the first dequeue also removes an item

Figure 2: The alternative design where the maxSize is not doubled. It shows the weaknesses outlined below: of requiring dequeue() to be called to utilise the entire space of the array; as well as never being able to store the maxSize number of elements at one time.

However, in implementation, it meant I needed to add a currentSize attribute to arbitrarily reject additional enqueue calls when the current size reached the maximum queue size. This is needed because if the input stack is filled and then an element dequeued, the queue only shrinks by one element, yet the input stack is large enough to add more than one new element. However, I decided this was better than requiring the user of the data structure to be calling dequeue often enough to make use of the entire queue – this may be useful if the queue is storing objects for short amounts of time as it would save space but would be much harder to understand.

Testing

For testing, I wanted to make sure the implementation of my design worked as expected for all potential inputs. I began by testing my array double stack and stack implementations as the queue depends upon their functionality.

After checking the factory can call a suitable constructor with a maximum size attribute, I wanted to make sure than an exception is thrown when a negative maxSize is given as you cannot create arrays with negative sizes.

Then, I needed to make sure the double stack could call suitable constructors to create and retrieve the first and second stacks, and then that those stacks are initialised correctly as empty stacks.

With the correctly initialised double stack and stacks, I could then check that elements can be added and retrieved from the first and second stacks. I needed to check both for each test as they use different starting points in the same array, and so wanted to make sure they could both be pushed to, popped from, filled, and emptied without impacting each other, or causing errors.

As my design stated, even maxSize values should split evenly and so I made checks that both could fill to exactly half the even maxSize value, and threw the correct exception when attempting to push more. For odd maxSize values, I made sure that each stack could hold half the maxSize when decremented. The example used was a maxSize of 9, which when decremented gives a total array size of 8, and so I checked that each stack could hold 4 elements each, and would throw an exception when the 5th was pushed on.

After checking the values could be added, I could then check that the size and isEmpty functions worked once items were added.

I then implemented the top method next because I could get the top element here when popping an item from the stacks. I added tests to make sure exceptions are thrown when the stack is empty, and full.

With this, I implemented tests for and implemented the pop functionality. I added checks for popping off empty and full stacks, and made sure that the elements are popped off in the first-in-last-out order which is expected for stacks. I then made tests check that the size of the stack is decremented when elements are popped off, and that when popped till empty, the isEmpty method returns the correct value.

Now, we should have all the functionality to interact with the stack and so I made sure that both the first and second stacks of a double stack could be filled at the same time without interfering with each other to make sure the topPosition indexing was working correctly.

I could then add tests to make sure the clear method resets the stacks when not-empty, full, and empty. This relies upon the popping functionality and so needed to be implemented last.

Finally, I could add tests to check that maxSize values of zero produce valid double stacks with

For the double stack queue, I then added very similar tests. I checked that negative maxSize values are rejected, and that positive numbers give valid queue instances with good initial values.

I then added tests to make sure that elements could be enqueued, and would only allow the maxSize number of values to be enqueued at one time without calling dequeue. I could then check that even and odd maxSize values allow the exact maxSize number of enqueued elements at one time. I made checks to make sure the size of the queue incremented, as well as the isEmpty value changing to false once one element is added.

I then added checks for dequeuing elements, with checking the correct first-in-first-out popping order of queues, and made sure that the size of the queue only decreased by one for each dequeue. Also, when the queue is reduced to size 0, that the isEmpty function returns true. Then, I could check that an exception is thrown when the total number of elements in the output and input stacks together is equal to the maxSize and an element is enqueued despite there being space in the input stack.

I could then add tests to make sure that the queue could be cleared, and the size and isEmpty functions returned expected values afterwards – essentially making sure that the clear method resets the queue to it’s initial state.

Finally, I could make sure that maxSize values of zero produce valid queues which throw exceptions on enqueuing and dequeueing, and will always return zero size and true to isEmpty.