



Queue (Implementation)

Lets look at the basic functionality and implementation of queues in Python!

We'll cover the following



- Implementation of Queues
 - Adding Helper Functions
- Complexities of Queue Operations

Implementation of Queues

Queues are implemented in many ways. They can be represented by using lists, Linked Lists, or even Stacks. But most commonly lists are used as the easiest way to implement Queues.

With typical arrays, however, the time complexity is $O(n)$ when dequeuing an element from the beginning of the queue. This is because when an element is removed, the addresses of all the subsequent elements must be shifted by 1, which makes it less efficient. With linked lists and doubly linked lists, the operations become faster.

Here, we will use a doubly-linked list to implement queues.

As discussed in the previous lesson, a typical Queue must contain the following standard methods:

- `enqueue(element)`
- `dequeue()`



- `is_empty()`
- `front()`
- `rear()`



We will take a look at these functions individually, but, before that, let's construct a class of Queue called `MyQueue` and create an object of this class with `queue_obj` name. The class consists of relevant functions for the queue and a data member called `items`. The data member is a doubly-linked list that holds all the elements in the queue. The code given below shows how to construct a Queue class.



```
main.py
DoublyLinkedList.py

1  from DoublyLinkedList import DoublyLinkedList
2
3  class MyQueue:
4      def __init__(self):
5          self.items = DoublyLinkedList()
6
7  queue_obj = MyQueue()
```

Adding Helper Functions

Now, before adding the `enqueue(element)` and `dequeue()` functions into this class, we need to implement some helper functions to keep the code simple and understandable. Here's the list of the helper functions that we will implement in the code below:



- `is_empty()`
- `front()`
- `rear()`
- `size()`



 Queue.py

main.py

DoublyLinkedList.py

```
from DoublyLinkedList import DoublyLinkedList

class MyQueue:
    def __init__(self):
        self.items = DoublyLinkedList()

    def is_empty(self):
        return self.items.length == 0

    def front(self):
        if self.is_empty():
            return None
        return self.items.get_head()

    def rear(self):
        if self.is_empty():
            return None
        return self.items.tail_node()

    def size(self):
        return self.items.length

if __name__ == "__main__" :
    queue_obj = MyQueue()

    print("is_empty(): " + str(queue_obj.is_empty()))
    print("rear(): " + str(queue_obj.rear()))
    print("front(): " + str(queue_obj.front()))
    print("size(): " + str(queue_obj.size()))
```





At the moment, `is_empty()` should return `True` and `front()` `None` because the queue is empty. We consider the last element of the list to be the `rear` (which means we will *enqueue* elements here!) and the first element to be the `front` (we will *dequeue* elements from here). You can also do the opposite - this is just how we are implementing queues in this course.

Now, examine the following extended code with the `enqueue(element)` and `dequeue()` functions added to the `MyQueue` class. We have created an object of the `MyQueue` class and will try to add and remove some elements from this queue by using these two functions. Let's try!



main.py

DoublyLinkedList.py

```
from DoublyLinkedList import DoublyLinkedList

class MyQueue:
    def __init__(self):
        self.items = DoublyLinkedList()

    def is_empty(self):
        return self.items.length == 0

    def front(self):
        if self.is_empty():
            return None
        return self.items.get_head()

    def rear(self):
        if self.is_empty():
            return None
        return self.items.tail_node()

    def size(self):
        return self.items.length

    def enqueue(self, value):
        return self.items.insert_tail(value)
```



```
def dequeue(self):
    return self.items.remove_head()

def print_list(self):
    return self.items.__str__()

if __name__ == "__main__" :
    queue_obj = MyQueue()
    print("queue.enqueue(2);")
    queue_obj.enqueue(2)
    print("queue.enqueue(4);")
    queue_obj.enqueue(4)
    print("queue.enqueue(6);")
    queue_obj.enqueue(6)
    print("queue.enqueue(8);")
    queue_obj.enqueue(8)
    print("queue.enqueue(10);")
    queue_obj.enqueue(10)

    queue_obj.print_list()

    print("is_empty(): " + str(queue_obj.is_empty()))
    print("front(): " + str(queue_obj.front()))
    print("rear(): " + str(queue_obj.rear()))
    print("size(): " + str(queue_obj.size()))
    print("Dequeue(): " + str(queue_obj.dequeue()))
    print("Dequeue(): " + str(queue_obj.dequeue()))
    print("queue.enqueue(12);")
    queue_obj.enqueue(12)
    print("queue.enqueue(14);")
    queue_obj.enqueue(14)

    while queue_obj.is_empty() is False:
        print("Dequeue(): " + str(queue_obj.dequeue()))

    print("is_empty(): " + str(queue_obj.is_empty()))
```



If you look at the output of the code, you can see that the elements are **enqueued** in the **rear** and **dequeued** from the **front**. This means that our queue works perfectly. Congratulations, you have now successfully implemented a Queue using a doubly-linked list!



Complexities of Queue Operations



Let's look at the time complexity of each queue operation.

Operation	Time Complexity
<code>is_empty()</code>	$O(1)$
<code>front()</code>	$O(1)$
<code>rear()</code>	$O(1)$
<code>size()</code>	$O(1)$
<code>enqueue(element)</code>	$O(1)$
<code>dequeue()</code>	$O(1)$

Now let's try some challenges that use your knowledge of stacks and queues. After that, we will take a look at some advanced data structures which are derived using the basic data structures that we have studied so far.

Interviewing soon? We've partnered with Hired so that companies apply to you instead of you applying to them. [See how](#) ⓘ



What is a Queue?


Challenge 1: Generate Binary Numbers...

?

📄

⚙️

☒ Completed

 Report an Issue

