1. Purpose and Efficacy of my Test Cases

My test cases specifically checked the string representations, sizes, and returned values after performing each of the methods for stacks, queues, and deques. Moreover, my test cases also checked my methods at various sizes including empty, size of 1(one value), and size of 3 (many values). Also, my test cases also checked that my methods worked when called 1 time and when it was called three times (multiple times). Constructing my test cases in this manner allowed me to thoroughly ensure that the methods in stacks, queues, and deques worked correctly.

1.1 Empty Stack / Queue / Deque Test Cases

The first thing I did when creating test cases for these data structures was to check their methods when the structure was empty. Furthermore, I checked if the size of each of these empty structures was initialized to 0 and that the empty string representation looked like this: '[ ]'. I tested the string representation and size when the structure was empty to ensure that I had properly defined what an empty stack, queue, and deque should be when created. After that, I created test cases to check that removing an element (pop for stacks, dequeue for queues, and pop front and back for deques) from the empty stack, queue, and deque ended up not changing the size nor string representation. The size should have remained 0 with the string representation still being '[ ]'. Also, I checked to see that removing an element from an empty stack, queue, and deque returned "None" intending to return nothing. Finally, I checked that the peek methods (peek for stacks and queues, peek front and back for deques) returned "None" in an empty stack, queue, and deque. I later discuss the problems associated with returning "None" when peeking or removing values from an empty data structure because of how it is different from returning nothing.

1.2 Push / Enqueue / Push Front & Back Test Cases

After creating my test cases for when these structures were empty, I checked all of their methods which added a new value into the structure (push for stacks, enqueue for queues, and push front and back for deques). First, I tested the string representation of these structures after adding one value into the structure and later adding three values into the structure. I tested adding one value to see if the method added the value into the structure and should have resulted in a string representation that looked like: '[value]'. Furthermore, I tested adding three values to test if these methods added the values in the correct order (adding a new value to the top of stacks, to the front of queues, and the front or back of deques depending on the method called) and should have resulted in a string representation that looked like: '[value1, value2, value3]' (the order value1, value2, and value3 varied depending on the data structure). After testing the string representation of these methods, I tested the size of these structures after adding one value and later adding three values. Moreover, I did so to test if the size was being updated correctly. After adding one value into the structure the size should have been updated to 1, and the size after adding three values should have been updated to 3.

1.3 Pop / Dequeue / Pop Front & Back Test Cases

Later on, I went to test the removal methods for these structures (pop for stacks, dequeue for queues, and pop front and back for deques). I first tested the returned values after removing one value from the structure. Furthermore, I tested these return values from structures that had one value-added into it and then with three values added into it before the removal. I tested the returned value of the removal methods after one value had been added to make sure that the removal methods returned a value that was in the structure. I also tested this after adding three values to check if the returned value was from the correct location in the structure (returning the value from the top of stacks, front of queues, and front or back queues depending on which pop was called). Next, I tested if the removal methods removed a value from the structure by checking its string representation after calling the removal method. Moreover, I tested this with structures that had one value-added in and then with three values added in before the removal. Immediately removing a value from a structure with only one value should have resulted in an empty string representation that was indicated by: '[ ]'. After that, I did the same thing but with 2 additional values, and I checked if the value had been removed from the correct location in the structure. After testing this, it should have printed out a string representation of the structure with only two values in it with the correct value being removed from the structure (removing the value from the top of stacks, front of queues, and front or back queues depending on which pop was called). Finally, I tested the size of each of the structures after removing a value. Moreover, I tested this by having one value-added into the structure then having three values added into the structure before the removal. Removing a value from a structure with only one value should have resulted in a size of 0 while removing a value from a structure with three values should have resulted in a size of 2. In addition to testing one removal call, I tested three removal method calls in cases when three values had already been added into the structure. I did this to make sure that multiple removal method calls worked.

1.4 Peek / Peek Front & Back Test Cases

Lastly, I tested the peek methods of these structures (peek for stacks/queues and peek front or back for deques). I first tested the return values of the peek methods for structures with one value-added into it and with three values added into it. Moreover, the test with one value tested if peek returned a value that was in the structure, and the test with three values was more for checking if it returned the value in the correct location in the structure (returning the value from the top of stacks, front of queues, and front or back queues depending on which peek was called). After that, I tested the string representations of the structures with one value-added into it and with three values added into it before calling the peek methods. The string representation after calling the peek method after adding in one value should have still been: '[value]'. Moreover, the string representation after calling the peek method with three values added should have still been '[value1, value2, value3]'. I tested this to make sure that calling the peek method did not change the structure in any way. Lastly, I tested the size of the structures with one value-added into it and with many values added into it before calling the peek methods. Similar to testing the string representation, I tested the size to make sure that the size of the structures did not change after calling the peek methods. Furthermore, the size of the structure with one value should have still been 1, and the size of the structure with three values should have still been 3.

1.5 Explanation of Deque test cases

For my pop and peek testing for deque, there were a lot more test cases than compared to my test cases in stack and queue. This was because deques can push values from the back/front, peek from the back/front, and pop from the back/front. I had to test all of the different interactions that could occur between these 6 different methods. For example, I had to test pop_front after pushing from the front and the back as well. This was necessary to check my Array_Deque and

Linked_List Deque implementations of a deque. These tests ensured that the front and back of my Array_Deque had been moved to their proper places depending on the method called. Also, for my Linked_List Deque, these tests made sure that my Linked_List methods were used correctly to create a deque.

2. Analysis of the Worst-case Asymptotic Performance of Methods in Stack/Queue/Deque

There were no performance differences between stacks, queues, and deques themselves. This was because the methods used in stacks and queues were inherited from deques. The difference in worst-case performance applied more to the two different types of deques that were created in this project (Array_Deque vs Linked_List_Deque).

2.1 String Methods for Stack/Queue/Deque

The string method for stacks, queue, and deques had a worse case performance time of O(n^2) when using Array_Deque. In my string method, I created a new string called "reordered_contents" which was intended to store the string representation of the structure. A for loop was used to concatenate the string version of each value in the structure into the reordered_contents. Moreover, the for loop and the string concatenation that was in the for loop were both O(n) which ended up making the overall worst-case performance of the string method using Array_Deque O(n^2).

```
def __str__(self):
   if self.__size == 0:
      return ('[ ]')
   reordered_contents = '[ '
   index = self.__front
   for i in range(self.__size):
      if index == self.__capacity:
         index = 0
      if i == self.__size - 1:
            reordered_contents = reordered_contents + str(self.__contents[index]) + ' ]'
            break
        reordered_contents = reordered_contents + str(self.__contents[index]) + ', '
        index+=1
    return reordered_contents
```

Figure 2.1.1: String method for Array_Deque

Similarly to Array_Deque, The string method for these structures using Linked_List_Deque had a worst-case performance of O(n^2) as well. This was because similar to Array_Deque, Linked_List_Deque also needed a for loop to go through each value in the structure and concatenate the string version of each value into a new string (new string is named "list" in my implementation).

```
def __str__(self):
    if self.__size == 0:
        return '[ ]'
    list = '[ '
    cur = self.__header
    for i in range(len(self)):
        if i == len(self) - 1:
            list = list + str(cur.next.val) + ' ]'
            break
        list = list + str(cur.next.val) + ', '
        cur = cur.next
    return list
```

Figure 2.1.2: String method for Linked_List (Linked_List_Deque string method returns the string method from Linked_List)

2.2 Length Methods for Stack/Queue/Deque

The length method for these structures had a worst-case performance of O(1) for Array_Deque as well as for Linked_List_Deque. In both implementations, all that needed to be done was to return the size of the deque which was constant time. In both my deques, I created a size variable that was initialized to 0 to keep track of the size of the structure (size increased by one whenever a new value was added and the size decreased by one whenever a value was removed).

```
def __len__(self):
  return self.__size
```

Figure 2.2.1: Length method for Array_Deque

```
def __len__(self):
  return len(self.__list)
```

Figure 2.2.2: Length method for Linked_List_Deque

2.3 Push/Enqueue/Push Front & Back for Stacks/Queues/Deques

The push, enqueue, and push front/back methods for these structures had a worse case performance of O(n) when using Array_Deque. This is because in the worst case, the push front/back methods in Array_Deque need to use the grow method. Moreover, the grow method is O(n) because a for loop is required to visit each value and put each value into a new list that is two times greater. Furthermore, my implementation of the push back method for Array_Deque is slightly different from its push front method. This is because the back of the deque never wraps back around to the front of the deque. This is because once the deque is full, the grow method is called giving more space for the back to keep increasing without having to wrap around to the front of the deque.

```
def __grow(self):
  self.__capacity = self.__capacity * 2
  new_contents = [None] * self.__capacity
  index = 0
  for i in range(self.__front, self.__front + self.__size):
    new_contents[index] = self.__contents[i % self.__size]
    index+=1
  self.__contents = new_contents
  self.__front = 0
  self.__back = self.__size - 1
```

```
        return self.__contents
```

Figure 2.3.1: Grow method for Array_Deque

```
    def push_front(self, val):
        if self.__size == self.__capacity:
            self.__grow()
        if self.__size == 0:
            self.__front = 0
            self.__back = 0
            self.__contents[self.__front] = val
            self.__size += 1
        elif self.__front == 0:
            self.__front = self.__capacity - 1
            self.__contents[self.__front] = val
            self.__size += 1
        else:
            self.__front -= 1
            self.__contents[self.__front] = val
            self.__size += 1
```

Figure 2.3.2: Push front method for Array_Deque

```
    def push_back(self, val):
        if self.__size == self.__capacity:
            self.__grow()
        if self.__size == 0:
            self.__front = 0
            self.__back = 0
            self.__contents[self.__front] = val
            self.__size += 1
        else:
            self.__back += 1
            self.__contents[self.__back] = val
            self.__size += 1
```

Figure 2.3.3: Push back method for Array_Deque

On the other hand, the push, enqueue, and push front/back methods for these structures had a worst-case performance of O(1) when using Linked_List_Deque. This occurred because the values were always added to the ends of the structure. Moreover, this made it so that the structure did not need to be traversed when insert_element_at was called from Linked_List which would have normally had a worst-case performance of O(n).

```
def push_front(self, val):
   if len(self.__list) == 0:
      self.__list.append_element(val)
   else:
      self.__list.insert_element_at(val, 0)
```

Figure 2.3.4: Push front method for Linked_List_Deque

```
def push_back(self, val):
   self.__list.append_element(val)
```

Figure 2.3.5: Push front method for Linked_List_Deque

2.4 Pop/Dequeue/Pop front & back Methods for Stacks/Queues/Deques

The pop, dequeue, and pop front/back methods for these structures when using Array_Deque had a worst-case performance of O(1). This was because when calling the pop front/back methods from Array_Deque, the value that needed to be popped could be removed without having to traverse through the list.

```
def pop_front(self):
   if self.__size == 0:
      return
   temp = self.__contents[self.__front]
   if self.__size == 1:
      self.__front = None
      self.__back = None
      self.__size = 0
      return temp
   self.__front += 1
   self.__size -= 1
   if self.__front == self.__capacity:
      self.__front = 0
   return temp
```

Figure 2.4.1: Pop front method for Array_Deque

```
def pop_back(self):
   if self.__size == 0:
      return
   temp = self.__contents[self.__back]
   if self.__size == 1:
      self.__front = None
      self.__back = None
```

```
            self.__size = 0
            return temp
        self.__back -= 1
        self.__size -= 1
        if self.__back < 0:
            self.__back = self.__capacity
        return temp
```

Figure 2.4.2: Pop back method for Array_Deque

The worst-case performance for pop front and back when using Linked_List_Deque was also O(1). Even though it used the remove_element_at method from Linked_List which was normally O(n), in this case, it was O(1). This was because the value that was removed was always on the ends of the structure. This meant that Linked_List_Deque never needed to be traversed which was what made the pop front and back method O(1).

```
    def pop_front(self):
        if len(self.__list) == 0:
            return
        return self.__list.remove_element_at(0)
```

Figure 2.4.3: Pop front method for Linked_List_Deque

```
    def pop_back(self):
        if len(self.__list) == 0:
            return
        return self.__list.remove_element_at(len(self.__list) - 1)
```

Figure 2.4.4: Pop back method for Linked_List_Deque

2.5 Peek/Peek front & back Methods for Stacks/Queues/Deques

The peek methods for these structures using Array_Deque had a worst-case performance of O(1). For Array_Deque, all that needed to be done was to return the value at index front or back depending on which peek method was called.

```
def peek_front(self):
   if self.__size == 0:
      return None
   return self.__contents[self.__front]
```

Figure 2.5.1: Peek front method for Array_Deque

```
def pop_back(self):
   if self.__size == 0:
      return
   temp = self.__contents[self.__back]
   if self.__size == 1:
      self.__front = None
      self.__back = None
      self.__size = 0
      return temp
   self.__back -= 1
   self.__size -= 1
   if self.__back < 0:
      self.__back = self.__capacity
   return temp
```

Figure 2.5.2: Peek back method for Array_Deque

For Linked_List_Deque, it used the get_element_at the method from Linked_List without traversing through a for loop. Furthermore, the method get_element_at never had to loop through the values because the values that needed to be returned were always on the ends of the structure making its worst-case performance O(1) as well.

```
def peek_front(self):
   if len(self.__list) == 0:
      return
   return self.__list.get_element_at(0)
```

Figure 2.5.3: Peek front method for Linked_List_Deque

```
def peek_back(self):
   if len(self.__list) == 0:
      return
```

```
        return self.__list.get_element_at(len(self.__list) - 1)
```

Figure 2.5.4: Peek back method for Linked_List_Deque

3. Performance Observations for Tower of Hanoi

From what I observed in the tower of Hanoi when changing the number of discs (n), the time doubled every time n increased by one. This meant that the time taken to complete the Hanoi method should be (2^n). I tested this doubling time by calling the Hanoi method at n = 1, 2, 3, 4, and 5. Then I ran each of the sizes for a total of 5 times to find the average time for each size and placed them in a table. From this table of average times, I created a graph with the number of discs (n) on the x-axis and average time (s) on the y-axis. The table and graph showed this 2^n relationship.

number of discs (n)	avg times (s)	#1	#2	#3	#4	#5
1	9.84E-05	9.07E-05	9.25E-05	0.00011	0.000109	8.91E-05
2	0.000177	0.000181	0.000176	0.000178	0.000172	0.000176
3	0.000347	0.000336	0.000355	0.00031	0.000416	0.000315
4	0.000655	0.000622	0.00064	0.000685	0.000636	0.000691
5	0.001375	0.001304	0.001404	0.00139	0.001323	0.001452

Figure 3.1: Table of the number of discs (n) vs. avg times (s)

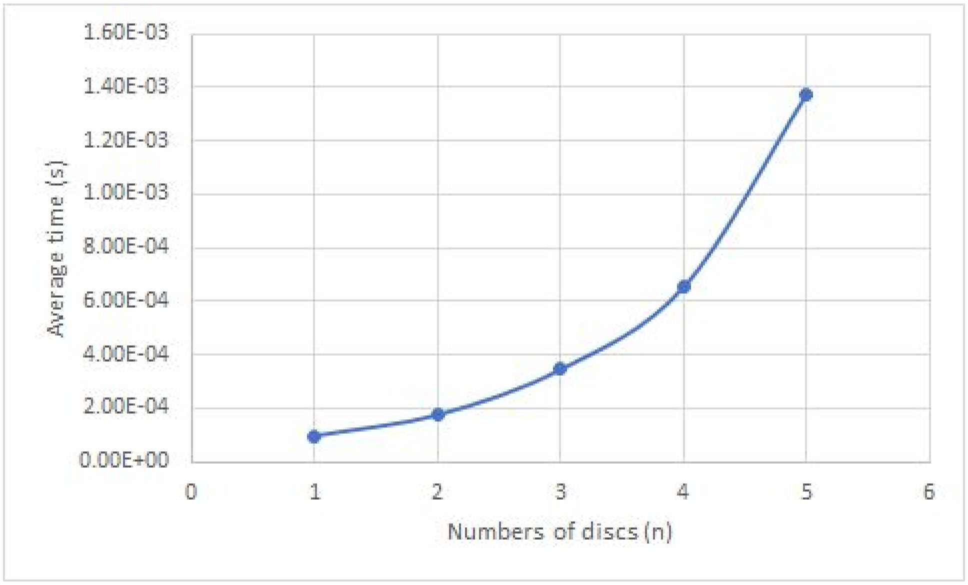

Figure 3.2: Graph of the number of discs (n) vs. avg times (s)

4. raising exceptions for pop and peek methods

In my implementation of the pop and peek methods, the value “None” was returned when pop or peek was called using an empty stack, queue, or deque. However, I intended to have the method return nothing, not return “None” which seemed almost the same as nothing at first glance. However, the difference between the two could be seen especially when used with the addition methods (push/enqueue/push front & back). For example, if the user tried to do something like deque.push_front(pop_front), the deque should be empty still. However, the value “None” would end up getting pushed into the deque. This would cause problems like increasing the size of the deque by one which should not be the case when nothing is added to an empty deque. Raising exceptions would prevent issues like this from occurring and prevent the user from accidentally misusing these structures.