Performance Analysis of the Methods in *Deque, Stack, Queue* classes

Comparison of two ways of implementing Array_Deque class:

I have considered two different ways of implementing an array deque: one is to keep track of two attributes, a reference to the front position and the size of the deque, and to initialize the front position at point 0 in an empty deque; the other is to keep three attributes, a reference to the front, a reference to the back and a size, and in this way I initialize the front and back reference both to be None in an empty deque.

Obviously, the consistency is sacrificed in the first implementation, because the front reference is supposed to point to the actual front position in the deque that has a value in it, while in my first implementation, the front references actually points to the position after the actual front position when the deque is empty initially or reduced to be empty in operations. The motivation I attempted to implement the deque this way is that by not setting the front reference to be None, I need not to add an *if* statement in every push method to catch the situation where the reference is at None and the reference arithmetic cannot be validly executed. However, I wrote both implementations down and compared their simplicity, and I don't find the first one significantly better:

First implementation:

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

Second implementation:

```
def push_front(self, val):
    if self.__size == self.__capacity:
        self.__grow()
    if self.__front is None:
        self.__front = 0
        self.__back = 0
    else:
        self.__front = (self.__front - 1 + self.__capacity) % self.__capacity
    self.__contents[self.__front] = val
    self.__size = self.__size + 1
```

Both implementations can push the value to the right place. The difference is tiny: the *if* statement in my second implementation is very simple. So, I don't think it's worthy of

sacrificing the consistency of my structure. Also, the difference between keeping a reference to the back position or not is not significant: in the first case one more step should be added to each push and pop method, while in the second case an arithmetic expression, which is slightly more costly should be added in the methods that involves the back position to find that back position. Since the first approach better carries out the beauty of a consistent data structure, I chose to use it.

Therefore, after examination, my Array_Deque finally prefers the second implementation, that is, I kept three attributes, front, back and size, and initialized front and back both to be None.

Performance Analysis of Deque methods:

__init__(self):

The worst-case performance of the constructer of implementations of *Deque* abstract base class is constant time *O(1).*

In the Array-based implementation, the work done by the constructer is fixed and there's no possible input value scale. The constructer first initializes an attribute *self.__capacity* to be 1, and then initializes an empty array *self.__contents* of the size of *self.__capacity;* then the constructer initialized two reference attribute, *self.__front* and *self.__back* both to point to None, and it finally initializes an attribute *self.__size* to be 0. All the works above are done in Python in constant steps. Therefore, the total performance time is a constant *O(1).*

Also, in the Linked List-based implementation, the constructer does constant-time work which is just creating a linked_list object to store the data. Since it has been shown in the writeup of project2 that constructing a linked_list object in my implementation is constant time, this deque implementation's constructer is also constant time.

__str__(self):

The worst-case performance of the *__str__()* method of the implementations of *Deque* abstract base class is linear time *O(n).*

In the Array-based implementation, the *__str__()* method has to loop *(self.__size - 1)* times to see and get every value stored in the deque (because the loop ends at one position before the back position, for the string operation done to the last element is different in that "," should not be appended after the last element), which is a function of *n.* Given that inside each loop the work is constant, only involving an index arithmetic to locate the index and a string conjunction operation, which are both able to be executed in constant time, the total performance of *__str__()* method is a function of *n* times a constant function plus a constant function that represents the remaining work of formatting the string. Therefore, the total performance is a function of *n,* which is linear time *O(n).*

In the Linked List-based implementation, the *__str__()* method just calls the *__str__()* method in the Linked_List class, which has been shown to be linear time *O(n),* and it basically does the same work as looping through the whole list, doing constant work inside

each loop, and doing constant work for the remaining part of the method. Therefore, the worst-case performance of *__str__()* method in a Linked_List-based deque implementation is linear time *O(n)*.

__len__(self):
The worst-case performance of the *__len__()* method in the implementations of deque class is constant time *O(1)*.
In the Array-based implementation, *__len__()* method just returns an instance variable, *self.__size,* without doing any count or loop. Since *self.__size* is kept as an attribute and updated every time the length of the deque is changed, in this particular method it is just constant time to look and return a value that is always stored at hand.
In the Linked_List-based implementation, the *__len__()* method just calls the *__len__()* method in the Linked_List class, which I have shown in the previous writeup to be constant time for the same reasons.

__grow(self)(exclusively array-based implementation)
The asymptotic performance of *__grow()* method in an Array-based implementation of deque class is linear time *O(n)*. The linear part of this method is that it has to loop through every values in the current *self.__contents* and pass them to the *new_contents,* which is a loop of *n* times. Since passing a value of one cell in an array to another cell in another array is a constant-time work, the total performance of this passing operation is a function of *n,* which is linear. Apart from this part, the remaining work is all constant time: constructing an empty array of twice length as the existing one, pointing the attribute *self.__contents* to the newly constructed array, and updating the *self.__capcacity.* Therefore, the total performance of the method is also a function of *n,* which is linear time *O(n),*

push_front(self, val):
The asymptotic performance of this method is different in the two implementations of deque class.
In the Array-based implementation, the worst-case asymptotic performance of the *push_front()* method is linear time *O(n)*. This is because of the *__grow()* method that is called inside the *push_front()* method. In the worst case, this linear time *__grow()* method is executed, making the total performance of the *push_front()* method a function of *n* plus a constant function which is the constant work of adding the value as the new front. However, the amortized performance of the *push_front()* method in and array-based deque class is constant time *O(1),* because a consecutive *n* calls of *push_front()* in an array-based deque should run at total *O(n)* times. If it is reasonable to assume there's no need to frequently resize the array, i.e., if the array-based deque is expected to remain at a moderate length (e.g, every *push_front()* is preceded by a *pop* so that the *push* will always run without calling the linear *__grow* method), the performance of *push_front()* can be even more efficient. The other work of advancing the reference of front circularly through the array

and assign the value to that new front position are all constant time *O(1)*.
In the Linked-List-based deque implementation, the worst-case asymptotic performance of the *push_front()* method is constant time *O(1)*. In this implementation, either the list is already empty, so the *push_front()* method calls an *append_element(val)* method in the linked list class, or the list is not empty, and an *insert_element_at(val,0)* is called. Both methods are constant time *O(1)* in a linked list class because a linked list supports constant-time access to its header and trailer attribute that can be used to conveniently modify the head and tail position. Therefore, the total performance of *push_front()* method in a linked_list-based deque is always constant time *O(1)*.

Comparison of linked-list push() and array push():
It seems that a linked_list-based implementation outperforms the array-based implementation, since the worst-case performance of the former is linear time *O(1)* while that of the latter is linear time *O(n)*. However, if the *__grow()* is not called in an array-based implementation, then the *push_front()* in an array is slightly faster and uses less memory space than a linked list *push_front()*. This is because for a linked list to insert or append element, a Node object has to be created and stored first, which cost memory and time, and there are more references needed to be set (the prev, next of the constructed Node, the prev of the Node after that constructed Node, and the next of the previous one). Whereas in an array, at most two reference has to be modified (front and back, in the case where they're initially None). Therefore, since the amortized performance of two implementations are the same, and the array-based *push_front()* may have slightly smaller coefficient and better memory use, it is reasonable to prefer an array-based deque when there the deque is not expected to grow frequently. On the other hand, if the deque is expected to go very long, the demerit of *__grow()* methods would outweigh the advantage of array implementation, and it is better to use a linked list-based deque. Furthermore, if the expected size of the deque is relatively small, and the number of *pushing* operations are also small, then it is also reasonable to prefer a linked-list-based deque because in a small number of operations, the linearity of *__grow()* would take up a large proportion, making the total performance of array-based deque inefficient.
Therefore, the principle of determining which implementation to use should be this: **if** the deque for my project is implemented by an array, what is the proportion of *__grow* methods that are executed to the total times of operations in my deque? If this proportion is large, as in the case of a very short deque and a continuously growing deque, then the array-deque is outperformed by linked list-deque in efficiency; while if this proportion is expected to be small, then the array-based deque should be preferred for slightly faster performance and great improvement in memory use.

pop_front(self)
The worst-case asymptotic performance of *pop_front()* method in the two implementations of deque abstract base class is the same constant time *O(1)*.

In an array-based implementation, the *pop_front()* method does three things: preserve the value at the current front position, make the front reference "-1", according to my selection of front direction, and decrement the size. These works can all be done in constant times because they only involve directly modifying or accessing the attributes. Further, the *if* statement catches the worst case in this method, i.e., the size goes to 0 after popping. In this case, two more constant-time work is done to set the front and back reference to be None. Therefore, the total performance of this method in array-based deque is a sum of constant functions, which is also constant time *O(1)*.
In a linked-list-based implementation, the worst-case performance of *pop_front()* method is also constant time *O(1),* because in the worst case, the *remove_element_at(0)* is called, which is constant time in a linked list class because the loop is not executed.
For the same reason as above, the array-based implementation is slightly faster and uses less memory space than linked list, namely that the array involves less modification of references and doesn't need extra space to store a Node object.

peek_front(val)
The worst-case performance of *peek_front()* method in both implementations of deque abstract base class is constant time *O(1)*.
In the array-based implementation, the work is just to return the value in *self.__contents* at the position of *self.__front.* Since both attributes are kept track of in the class, this work is constant time *O(1)*.
In the linked-list-based deque implementation, when the deque is not empty, which is the worst case, *get_element_at(0)* is called. This method in a linked list class is constant time as has been shown in the previous writeup, because linked list supports constant-time access to its head position by *header.next.* Therefore, the *peek_front()* is constant time *O(1)*.
For the same reason, the array-based implementation uses less memory space than linked list, namely that the array doesn't need extra space to store a Node object.

push/pop/peek_back()
The worst-case asymptotic performance and performance feature of *push/pop/peek_back()* methods are exactly the same as *push/pop/peek_front()*. This is justified because in the array-based implementation, I have both the front and back reference stored as attributes, so modifying both ends can follow the same procedure; and in a linked list, operations on the head position and tail position are equally efficient because of the constant-time access to both positions.

Performance Analysis of Stack Methods

__init__(self)
The worst-case performance of the constructer of *Stack* class is constant time *O(1)*. The constructer of *Stack* class calls a function *get_deque()*. This function either returns an array-

deque object or returns a linked-list-deque object by calling their respective constructer, depending on the parameter sent to it. Since it has been shown above that either of the constructers of array-deque or linked list-deque is constant time. The performance of *get_deque()* function is constant time *O(1)*. Therefore, the performance of the constructer is also constant time *O(1)*.

__str__(self)
The worst-case performance of the *__str__()* method is linear time *O(n)*. Since the *__str__()* method in *Stack* class just calls the *__str__()* method in the deque abstract base class, its performance should be identical to that in the deque class; and since it has been shown that the *__str__()* method in the deque class is linear time no matter which implementation is chosen, the *__str__()* method in the *Stack* class is also linear time no matter it is implemented with an array-deque or a linked list-deque.

__len__(self)
For the same reason as above, the worst-case performance of *__len__()* method in a *Stack* class is the same as the performance of *__len__()* method in either deque implementation, which is constant time *O(1)*.

push(self, val)
Since the *push()* method in the *Stack* class calls the *push_front()* method in either implementation of the deque abstract base class, the performance feature of this method is identical to that in Array-deque or Linked list-deque, which I have discussed above.
To briefly recap, if the Stack is implemented with an array, then the worst-case asymptotic performance of the *push()* method is linear time *O(n)* because of the linear *__grow()* method. However, if it is reasonable to suppose the *__grow()* is not frequently executed, then the amortized performance of *push()* method is constant time *O(1)*.
On the other hand, if the *Stack* is implemented with a linked list-deque, then the worst-case performance of the *push()* method is constant time *O(n)* with slighter bigger coefficient and more memory use than the array-*push* without the *__grow()* method.

pop(self)
From the same line of reasoning, the performance feature of *pop()* method in the *Stack* class is identical to one of the *pop_front()* method of the two implementations of the deque abstract base class. In this case, the worst-case performance of *pop_front()* method in either array-deque or linked list-deque is constant time *O(1)*. Therefore, the worst-case performance of *pop()* method in a *Stack* class is constant time *O(1)*.

peek()
From the same line of reasoning, the worst-case performance of *peek()* method in the *Stack* class is constant time *O(1)* because in either an array-based deque or a linked list-based

deque, the *peek_front()* method cost constant time *O(1)*.

Performance Analysis of Queue Methods

__init__(self)
The worst-case performance of the constructer of *Queue* class is constant time *O(1),* which is the same as that of *Stack* constructer. The two constructers do exactly the same work as calling the *get_deque()* function and use the returned deque from that function as an instance variable. Therefore, their performance feature is also identical, which is worst-case *O(1)*.

__str__(self)
The worst-case performance of the *__str__()* method in the *Queue* class is linear time *O(n)*. Since this *__str__()* method calls the *__str__()* method in one of the implementations of the deque abstract base class, the work done by it is identical to the work of one of the implementation's *__str__()* method depending on the return value of *get_deque* function. Given that the array-deque and the linked list-deque both have a linear worst-case performance *__str__()* method, the *__str__()* method in the *Queue* class which uses either of them must also be linear time *O(n)*.

__len__(self)
The worst-case performance of the *__len__()* method in the *Queue* class is constant time *O(1)*. The reason is the same as that of *__str__()* method: the *__len__()* method in the *Queue* class calls the *__len__()* method in either of the two implementations of the deque abstract base class, and both of them are constant time as have been shown above. Therefore, no matter which deque is used, the *__len__()* method in the *Queue* costs constant time *O(1)*.

enqueue(self, val)
The performance feature of the *enqueue()* method is similar to the *push()* method in the *Stack* class, since the *push()* calls the *push_front()* in either of the two deque implementations and the *enqueue()* calls *push_back(),* and the performance feature of *push_front()* and *push_back()* in a deque implementation has been shown to be similar.
Therefore, if the *Queue* class uses an array-deque as its instance variable, then the worst-case asymptotic performance of the *enqueue()* method is linear time *O(n)* because of the linear *__grow()* method. However, if there's good reasons to expect the *__grow()* function is not frequently executed as opposed to the total number of *enqueue(),* then the amortized performance is constant time *O(1)*, and the memory use is relatively small.
On the other hand, if the *Queue* uses a linked list-deque to store data, then the asymptotic performance of the *enqueue()* method is constant time *O(1),* but with slightly higher coefficient than the array-based *enqueue* without executing the *__grow(),* and it cost more

memory space to store the created Node object.

dequeue()

The worst-case performance of the *dequeue()* method is the same as that of the *pop()* method in the *Stack* class, which is constant time *O(1)*. The reason is the same: it calls the *pop_front()* method in either of the two deque implementations that is used to store data, and either of them has worst-case constant time performance, which has been shown above. Therefore no matter which one is used, the *dequeue()* method in the *Queue* class is constant time *O(1)*.

Performance analysis of choosing front and back position

First of all, in my array-based implementation of the *deque* class, I choose the "-1" in the array cell operation to be the front direction, and "+1" to be the back direction, which means in pushing front and popping back, the front/back reference should "-1", and in pushing back and popping front, the back/front reference should "+1". Then, in the linked-list-based implementation of the *deque* class, I use the descriptive head position in the linked list as the front of my deque and the tail in the list as my back.

I want to clarify first that my choice of front and back position doesn't matter in the performance analysis of either array or linked list based deque.

In an array-deque, if I choose the front as the way I did, then in the *push_front()* method, the index arithmetic should be:

```
self.__front = (self.__front - 1 + self.__capacity) % self.__capacity
```

and in the *push_back()* method the arithmetic is:

```
self.__back = (self.__back + 1) % self.__capacity
```

It can be clearly seen that if I choose the reverse way as I did, then two arithmetic should reverse accordingly, but the overall performance is strictly the same since both front and back operation should be frequently called of a deque. The same goes for the two *pop()* method.

Further, in a linked list-based implementation, the choice of front and back doesn't matter either. In my choice, the *push_front()* method would have the *if* statement to catch the situation where the deque is empty:

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

and in the *push_back()* method there need not be such conditional:

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

If I choose the tail to be front and head to be back, then just the reverse happens: my

push_front() position would be straightforward but the *push_back()* have to have a conditional to handle the empty deque. And the overall performance of my deque structure is strictly the same.

For the implementation of *Stack* and *Queue,* I choose the original front position in the array-deque and the head position in the linked list-deque as their front and the corresponding back as their back. I do this way for the readability and consistency of my code, for in this way, the *push* and *pop* in the *Stack* corresponds to *push_front* and *pop_front,* which is consistent with the definition of a *Stack* and is easily understandable; and similarly, in a *Queue,* the *enqueue* and *dequeue* correspond to *push_back* and *pop_front,* which is also very consistent and easy to recognize.

I recognized some reasons for doing other wise, i.e., to simplify the *push* or *enqueue* so that they need not involve an *if* statement which catches the case where the list is empty and the linked list-deque *insert* method cannot be executed. I want to argue that given what we're expected to do, this makes no difference on the overall performance of the two classes, *Stack* and *Queue.* First, in a linked list-based deque implementation, no matter which position I set to be front, *head* or *tail,* one of the *push_front* and *push_back* has to have an *if* statement for the above use. If I set the *head* to be the front, then I have to consider the empty list case in my *push_front* method; on the other hand, if I set the *tail* to be front, then I have to consider the empty case when I call the *push_back* method. No matter how I choose the front, one method has to be conditional and slightly more complicated than the other, which is because of the nature of my linked-list class.

This being established, now consider the *Stack* and *Queue.* The *Stack* class needs *push_front* method while the *Queue* class needs *push_back* method. This means that no matter how I choose the front and back position, one of the *Stack* and *Queue* class has to have a push method that has an *if* conditional. The choice of front and back position matters when only a *Stack* or only a *Queue* is to be implemented, but not when both are implemented.

However, there is one way to escape my problem: to use reverse front in *Stack* and *Queue* class, i.e., in a *Stack* class, call *push_back* and *pop_back* while in the *Queue* class call *push_back and pop_front.* This way, the push method in the two classes are the same, so it is possible to choose the appropriate front to simplify the method. This, I argue, will cost greater trouble, which is with the string representation. Since the *__str__()* method in my deque implementations only supports representing the deque from front to back, if the *Stack* class has the front of the deque as its back and the back of the deque as its front, then it can no longer use the *__str__()* of the deque to represent its elements in string. If the *Stack* class has to loop through its element to have its own *__str__()* method, it will cause huge change to the whole structure and greatly undermine the consistency and simplicity of the whole deque-stack-queue structure. I don't think this cost is worth paying, since the performance improvement is really slight.

Purpose and Efficacy of Test Cases

Unit Test Cases for the Deque Class

In testing the *deque* class methods, I focus especially on testing the possible difficulties on the array-based implementation of the *deque* class. The linked list-based *deque* is antecedently more reliable because it doesn't build the structure from scratch, but just calls the corresponding methods in the linked list class which I have tested in the previous project. Due to the fact that a doubly linked list, if only considered of its most efficient operations, works very much like a deque in itself, I think the linked list-based *deque* should work fine as long as the linked list works fine.

For the array-based deque implementation, the structure of a deque is constructed within a circular array by keeping track of three attributes, front, back and size. Since every method that updates the array-deque would change some of these attributes and is possible to affects subsequent operations, the test of array-deque should explicitly include tests that ensures that a sequence of different operations, which involves different ways of updating the three attributes, can work fine. This kind of test should be added to the standard tests similar to that of the linked list class that test the efficacy of single method.

Therefore, following this principle of "standard one method test + deliberately designed test of a sequence of different methods", and based on the feature of a deque, I designed the following blocks of test code:

Testing *push_front()* and *push_back()*:

The first block of test code is to test that I can successfully update my deque by passing values to the front or back position. The success means three things: first, I should have a successfully updated deque which can be represented in the string in the right format; second, I should have a successfully updated size attribute, which reflects in the right output of *__len__()* method; third, I should have my front and back reference rightly updated so that following operations based on these references can work without causing problems. The third overlaps with the first to some degree, but it is a separate demand of my testing purpose because in principle it is possible for one of my *push* or *str* call to produce a rightly updated deque but causing problem in the reference somehow that the following different function call will fail. In this block I have the following test:

def test_empty_deque_string(self):
    this is to test the *__str__()* method can output the empty deque in the right format.

def test_get_empty_length(self):
    this is to test the *__len__()* method can output the length of an empy deque rightly.

def test_push_front_empty(self):
    this is to test the *push_front()* method can successfully update an empty string to a string with one element.

def test_get_one_length_front(self):
    this is to test the length of such updated deque is right.

def test_push_front_one(self):

this is to test the *push_front()* methodcan successfully add an element to a deque with one element at the front, and the updated deque is formatted rightly

def test_get_two_length_front(self):

this is to test the length of the updated deque by the above operation is right.

…

Note: I explicitly uses the *push* methods three times to eventually test the string format and length of a deque with 3 element; this is not necessary, but I want to do this because I want to have the *__grow()* method executed twice to see it works fine without causing problems.

def test_push_back_empty(self):

def test_push_back_one(self):

…

This is where I do the same test as above, but with the *push_back()* method to test the same efficacy of *push_back()* method on its own: updating the deque in the right way, and successfully updating the length.

```
def test_front_back(self):
    self.__deque.push_front('Data')
    self.__deque.push_back('Structure')
    self.assertEqual('[ Data, Structure ]', str(self.__deque))
def test_back_front(self):
    self.__deque.push_back('Data')
    self.__deque.push_front('Structure')
    self.assertEqual('[ Structure, Data ]', str(self.__deque))
```

After testing and knowing that the *push_front()* and *push_back()* methods are both efficacious in themselves, I add these two tests to see that they can work well after each other, i.e., the call of one method updates the front and back reference both in the right way that the following other method can work perfectly.

def test_get_two_length_front_back(self):

def test_get_two_length_back_front(self):

In these two methods I do the same thing as above, updating the deque using a consecutive *push_front(back)* and *push_back(front),* and see the length is updated rightly. This is not 100% necessary given that it has been shown that both *push_front()* and *push_back()* can update the length rightly. But I still do this test for consistency.

Testing *pop_front()* and *pop_back():*

In the second block of test code, I purpose to test the efficacy and reliability of my *pop*

methods. For each *pop* method in itself, there are three things to test: first, the return value is right; second, the remaining deque is right and the value is successfully popped out of the deque; third, the length is rightly updated. Besides the above, I want to show that my *pop* methods update the front and back reference in the right way that following operations can succeed. For this aim, I tested the efficacy of pushing values after popping, and see the deque and length are both rightly updated. I have the following test codes:

```
    def test_pop_front_empty_return_value(self):
    to_return = self.__deque.pop_front()
    self.assertEqual(None, to_return)

  def test_pop_front_empty_remaining_deque(self):
    to_return = self.__deque.pop_front()
    self.assertEqual('[ ]', str(self.__deque))

  def test_pop_front_empty_length(self):
    to_return = self.__deque.pop_front()
    self.assertEqual(0, len(self.__deque))

  def test_pop_back_empty_return_value(self):
  def test_pop_back_empty_remaining_deque(self):
  def test_pop_back_empty_length(self):
```

This is to explicitly show that *pop_front()* and *pop_back()* should have no effect on an already empty deque, i.e., no value is returned, the deque is exactly unchanged, and the length is 0.

```
  def test_pop_front_leaving_empty_return_value(self):
  def test_pop_front_leaving_empty_remaining_deque(self):
  def test_pop_front_leaving_empty_length(self):
  …
  def test_pop_back_leaving_one_return_value(self):
  def test_pop_back_leaving_one_remaining_deque(self):
  def test_pop_back_leaving_one_length(self):
```

These tests are to show that *pop_front()* and *pop_back()* meets the three conditions above in updating nonempty deques.

```
  def test_pop_front_leaving_one_push_front(self):
        self.__deque.push_front('Data')
        self.__deque.push_front('Structure')
        to_return = self.__deque.pop_front()
```

```
        self.__deque.push_front('Ye')
        self.assertEqual('[ Ye, Data ]', str(self.__deque))

    def test_pop_front_leaving_one_push_back(self):
        self.__deque.push_front('Data')
        self.__deque.push_front('Structure')
        to_return = self.__deque.pop_front()
        self.__deque.push_back('Ye')
        self.assertEqual('[ Data, Ye ]', str(self.__deque))
```

Then, after making sure that *push_front()* and *push_back()* methods are both efficacious in themselves, I test to see that they're a part of a reliable structure, i.e., calling them can update the front and back reference in the right way that following work can be done without problem. For this purpose, I tested to push values after popping and see that the deque is updated in the right way and length is right.

```
    def test_pop_back_leaving_one_push_front(self):
    def test_pop_back_leaving_one_push_back(self):
```

This is to test the same thing with *pop_back()*

Testing *peek_front()* and *peek_back():*

In the third block of test I tested the *peek* methods that access and returns the front or back value without changing the deque. In this block of test, I don't have to worry about the impact of *peek* on further operations because *peek* methods are not expected to update any attribute; rather, I have to ensure that *peek* has absolutely no impact on any other operations. Therefore, what I have to test are three things: first, the return value is right; second, the deque is left exactly unchanged; third, the length is left unchanged. After testing these, I should add a few tests to show that any operations after *peek* can work well:

```
    def test_peek_front_empty_return_value(self):
    def test_peek_front_empty_remaining_deque(self):
    def test_peek_front_empty_length(self):
    …
```

These tests are to test that *peek_front()* meets the three conditions above, i.e., it returns the right value, leaves the deque unchanged and leaves the length unchanged. Also, *peek_front()* should have no effect on an empty deque: None is returned, the deque is still empty, and the length is still 0.

```
    def test_peek_back_empty_return_value(self):
    def test_peek_back_empty_remaining_deque(self):
```

```
def test_peek_back_empty_length(self):
…
```
These are to test the same thing with *peek_back()*

```
def test_peek_front_push_front_remaining_deque(self):
        self.__deque.push_front('Ye')
        to_return = self.__deque.peek_front()
        self.__deque.push_front('Yangchen')
        self.assertEqual('[ Yangchen, Ye ]', str(self.__deque))

def test_peek_front_push_front_length(self):
        self.__deque.push_front('Ye')
        to_return = self.__deque.peek_front()
        self.__deque.push_front('Yangchen')
        self.assertEqual(2, len(self.__deque))
…
```
Then, after testing the *peek_front()* and *peek_back()* both work well in themselves, I added these codes to show that they're capable of being integrated in a reliable structure, i.e., any subsequent deque operations can work successfully after calling *peek_front()*.

```
def test_peek_back_push_front_remaining_deque(self):
def test_peek_back_push_front_length(self):
…
```
These are to do the same test with *peek_back.*

Unit Test Cases for the Stack and Queue Classes

The test cases for *Stack* and *Queue* classes are following the same logic for testing *deque* class, and they need not be so detailed and tedious as in *deque* class tests. After all, the *Stack* and *Queue* methods are just a subset of the methods in *deque* class, and once we have ensured that all methods in the *deque* class works well in themselves and as an integrated solid structure, the methods in *Stack* and *Queue* class should work well automatically. The specific purpose of my test codes about *Stack* and *Queue* classes is to see that the methods in these two classes correctly call the corresponding methods in the *deque* class. If this is not so, then I can clearly see that the corresponding method in the *deque* class passed the test while the method in the *Stack* or *Queue* class fails. This is the only possibility that my *Stack* and *Queue* classes can work badly once I have assured that my *deque* class works finely. Further, since in such cases the methods in *Stack* and *Queue* classes must fail the test on their own, I don't need to test a sequence of methods called one after another as I have done in testing the *deque* structure. Therefore, the test blocks I have for *Stack* and *Queues* are just duplicates of the blocks I have for *push* and *pop* in my

deque structure respectively, and I don't need to test the combined effects of the methods as a reliable whole structure.

The performance observation of Hanoi

My algorithm for solving the Hanoi problem is as follows:

```
def Hanoi(n):
    source = Stack()
    dest = Stack()
    aux = Stack()
    i = n-1
    while i >= 0:
        source.push(i)
        i = i - 1
    Hanoi_rec(n-1, source, aux, dest)

def Hanoi_rec(n, s, a, d):
    print(n, s, a, d)
    # TODO replace pass with your base and recursive cases.
    if n == 0:
        d.push(s.pop())
    else:
        Hanoi_rec(n-1, s, d, a)
        d.push(s.pop())
        Hanoi_rec(n-1, a, s, d)
    print(n, s, a, d)
    print()
```

It is a recursive function; and suppose the performance time of solving the Hanoi problem with *n* discs is T(n). Then it is observed that:

(1) T(1) = 1, since when *n=1,* the work is constant, just involving a *pop* and a *push* of two stacks, which have been shown to be constant time.

(2) T(n) = 2T(n-1) + 1, since to perform any Hanoi(n) involves performing two Hanoi(n-1) and a constant-time work of popping and pushing.

From (1) and (2) it can be seen that T(n) grows geometrically, and by observing that: T(1): 1, T(2): 3, T(3): 7, T(4): 15… I made the conjecture that T(n) = 2^n-1, which is exponential time.

My conjecture can be proved by mathematical induction:

Proof: T(n) = 2^n-1 holds for all n >= 1,

Step1: when n = 1, T(1) = 2^1-1 = 1 is true, therefore, T(n) = 2^n-1 is true when n=1.

Step2: assume that T(n) = 2^n-1 is true when n=k, k >= 1 i.e., T(k) = 2^k-1. Then, consider T(n+1); n+1 = k+1, T(k+1) = 2T(k) + 1 = 2*(2^k-1) + 1 = 2^{k+1}-1; which is true. Therefore, T(k+1) = 2^{k+1}-1 is true for all k >=1 as long as T(k) = 2^k-1.

Therefore, combining Step1 and Step2 by the principle of mathematical induction, it is proved that T(n) = 2^n-1.

Discussion of the decision not to raise exceptions

I think the decision not to raise exceptions in my *deque* implementations is appropriate, and actually is consistent with what an exception is expected to do.

I think the exception is usually used to catch unexpected input such that, if it enters the program, the program will confront the risk of crash. For example, in my linked list class, an IndexError exception is raised when my user tries to call *insert_element_at(val, -1)*. This is because I cannot let -1 enter my program as a parameter and be passed into my loop *for k in range(-1),* which is an unacceptable index. For similar reasons, I cannot let my user pass a value bigger than or equal to *len(list)* as a parameter into my program because that would cause my index to point to *self.__trailer*, or some place after it, which is undefined. In all above situations, I have urgent and compelling reasons to block an invalid operation **by users** through an exception, otherwise the consistency of my program will be undermined.

However, the situation is different in my *deque* implementations. In my *deque* implementation, there are no such invalid inputs that have to be blocked and informed to the users by an exception: the only input the users have to pass are values, which cannot be invalid. If there's no index, you cannot raise an *IndexError.* What my methods in my *deque* implementation has to do, i.e., check if the deque is empty before popping or peeking anything, are not interacting with user input, but just part of the methods' inner logic. I think in these cases it is appropriate to not raise exceptions and interrupt with the users' work flow. In other word, exception handles situations that are unexpected by the inner logic of the method, often posed by invalid user inputs, while in my *deque* classes, there're no **unexpected** situations, because "popping an empty deque should have no effect" is what has already been expected by the inner logic of pop methods. Therefore, I suggest not raising an exception for that. A similar case happens in the *rotate_left()* method of my linked list class. In that method, "rotating an empty list should have no effect" is something that has already been expected in the logic of the method, so no exception is raised to handle this situation.

Despite what being argued, the method of returning None in my functions instead of raising an exception has its drawbacks. The first one that comes into my mind is that a deque with one element "None" would be indistinguishable from an empty deque. All of

the *pop_front(), pop_back(), peek_front(), peek_back()* will return exactly the same value when called on both two deques. For this reason, when my user calls a *pop* method and finds that the returned value is None, he would not be able to determine what this really means: does this None mean that my deque is empty? Or does this mean the front/back value stored in my deque is None? This limitation would remain even if I change the return value to be anything, like a string "your deque is empty": as long as the return value is itself a valid value that can be stored in a deque, the two situations will be indistinguishable. However, since it is very unlikely that my user would use a deque to store a string "your deque is empty", this alternative might work better than the one I'm taking now; but to ultimately resolve this problem, I suggest I should raise a specific type of *Empty* exception, which is in its essence a different type of returning so the user cannot mistake the message for any possible value stored in the deque. This is similar to what