Anatoly Osgood
Project 3 Writeup

## Array Deque:

**__init__:**

I added front and back parameters which contain the index of both of these pieces of information. I also added size which tells me the size of the deque. The worst-case performance of this method is constant time.

**__str__:**

This method's worst-case performance is O(n) or linear time. If the size of the list is 0 it returns the empty brackets, and if the size of the list is 1 it returns the brackets with the value inside. Otherwise, it loops through every single element of the list except for the last one, adding a comma and a space after each value. Finally, it adds the last value on without a comma and adds the concluding bracket. The reason that this method performs in linear time is that it always loops n-1 values.

**__len__:**

This method works in constant time, it returns the value stored in the size variable.

**__grow:**

This method's worst-case performance is also O(n). I store the old contents of the original array in old_contents. Then I double capacity and create a new array filled with None's that has the new capacity. Then I create a variable called walk that stores the value of the index of the front of the old contents. Then I loop through the entirety of the old array and set the new array's cells (starting at index 0) to their corresponding values in the old contents by using walk to iterate through the old contents. This is done to copy the old contents over to the new array. I used modulo to go through the circular array, specifically using walk to do accomplish this. I make walk bigger by one every loop in order to go to the next cell and then mod it against the size so that it can wrap around if the back is to the left of the front of the array.

**push_front:**

This method's worst-case performance is also O(n). First, it checks if the size of the array has reached capacity. If it has, it calls the __grow method. This call to the __grow method is what makes the worst-case performance of this method O(n). The reason why this method is O(n) is above. After calling __grow, it adjusts the front pointer by subtracting 1, adding the capacity and then taking the mod of that against the capacity. This makes the front index go to the left. The mod helps because the array is circular and lets us navigate around the array. Then it adds the pushed value to the index where the new front pointer points. Finally, it increases the variable representing the size of the deque by 1.

**pop_front:**

This method's worst-case performance is constant time. First, I use temp_front to store the index of the value we are removing from the front of the deque. Then I add one to the value of front and take the mod against the capacity. This is because a pop to the front moves the index to the right. I decrement size by 1 because we are removing an element. Finally, I return the value that we are popping by using temp_front to access the value. This method works in constant time because no matter what this method works as described above and doesn't depend at all on the length on the deque.

**peek_front:**

This method's worst-case performance is constant time. If the deque is empty, then the method returns None. Otherwise, I use the front pointer to access the variable in the front position and return it. This method works in constant time because no matter what this method works as described above and doesn't depend at all on the length on the deque.

**push_back:**

This method is very similar to push_front and also has a worst-case performance of O(n). First, it checks if the size of the array has reached capacity. If it has, it calls the __grow method. This call to the __grow method is what makes the worst-case performance of this method O(n). The reason why this method is O(n) is above. After calling __grow, it adjusts the back pointer by adding 1 and then taking the mod of that against the capacity. This makes the back index go to the right. The mod helps because the array is circular and lets us navigate around the array. Then it adds the pushed value to the index where the new back pointer points. Finally, it increases the variable representing the size of the deque by 1.

**pop_back:**

This method's worst-case performance is constant time. First, I use temp_back to store the index of the value we are removing from the back of the deque. Then I subtract one from the value of back, add the capacity, and take the mod against the capacity. This is because a pop to the back moves the index to the left. I decrement size by 1 because we are removing an element. Finally, I return the value that we are popping by using temp_back to access the value. This method works in constant time because no matter what this method works as described above and doesn't depend at all on the length on the deque.

**peek_back:**

This method's worst-case performance is constant time. If the deque is empty, then the method returns None. Otherwise, I use the back pointer to access the variable in the back position and return it. This method works in constant time because no matter what this method works as described above and doesn't depend at all on the length on the deque.

## LinkedList Deque:

**original methods:**

The __init__, __str__, and __len__ methods can't be changed. The __init__ and __len__ methods are in constant time as they do not depend on any input nor the length of the deque. The __str__ method calls LinkedList's str method which works in O(n) as it must go through every number in the deque.

**push_front:**

First, it checks if the length of the deque is 0. If it is, it calls the LinkedList's append_element method as that is the only way to add a value to an empty LinkedList. This method adds a node to the tail position of the list and increases the size of the list by one accordingly. Importantly, this method is the only way of adding a node the tail position. First, the method creates a new node with a value equivalent to the value passed to it by the user, we'll call this node "newest". Then, it looks at the node in the current tail position, takes its next variable and sets it to point at newest. Then, the method takes newest's previous variable and sets it equal to the node that is in the tail position. After that, it sets newest's next variable and points it at trailer, while setting trailer's previous variable to point at newest. It then increments the size of list by 1 as 1 new node as has been appended in the tail position. The performance of this method is also O(1) or constant time.
Otherwise, it calls the LinkedList's insert_element_at method to add the value at position 0. This method adds a node at any index except for the tail index. This is because the append_element method is used for adding any nodes to the tail. If the user tries to add a node to the tail, an IndexError is raised. First, this method checks the index that is passed. If the index passed is greater than or equal to the size of the list or less than 0, an IndexError is raised. This is to avoid any incorrect indices or an attempt to add a node at the tail. If the index passed is in the second half of the list (indices greater than the size divided by 2), the method creates a new node with a value equivalent to the value passed to it by the user, we'll call this node "newest". Then, I created a variable called current which starts at the trailer position and walks backward to the node right after the index we need to insert the node. Then I set newest.previous equal to current.previous so that newest is between current and the node that used to before it. Then I

set newest.next to point at current and current.previous to point at newest. Finally, I pointed the node before newest (accessed by newest.previous) to point its next at newest. I then increment the size by one.
Otherwise, if the index is in the first half of the list, (less than or equal to the size divided by 2) the method creates a new node with a value equivalent to the value passed to it by the user, we'll call this node "newest". Then, I created a variable called current which starts at the header position and walks through the list the node right before the index where newest is to be inserted. Then it sets newest's next variable to point at the next node in the list (via current's next) and sets newest's previous variable to point at current. Then, the method sets current's next and points it at newest. Finally, I set the node after newest to use its previous to point at newest. I used newest.next to access the node ahead of it because, current.next now points at newest.
The worst-case performance of this method is constant time as the value is always inserted at index 0 which is accessed through the header. Therefore, it never has to change based on the size of list and always does the same thing - making it run in constant time.

**pop_front:**

If the length of the deque is 0 it returns None. Otherwise, it calls the LinkedList method remove_element_at.
This method removes and returns the value of the node at the index. The first thing I do is check the index value passed by the user. If the index passed is greater than or equal to the size of the list or less than 0, an IndexError is raised. This is to avoid any incorrect indices. If the index passed is in the second half of the list (indices greater than the size divided by 2), I create a variable called current at the trailer. It starts at the trailer position and walks backward to the node right after the index we need to remove the node. I then create a new variable called element which is the node I'd like to remove It is accessed by calling current.previous. I then set current.previous to element.previous and current.previous.next to current so that element is no longer pointed to by either node. Finally, I decrement the size by 1 and return the value of element as the spec requires.
Otherwise, if the index is in the first half of the list, (less than or equal to the size divided by 2), I again create a variable called current at the header. Just like in insert_element_at, I loop forward through the list so that current is equal to the node right before the element that the user would like to remove. Then I create a variable called element which is the node that the user would like to remove. Then, I set current.next equal to the node after the node that is removed. This accessed by using element.next. Then I set that node's previous value to point at current by using current.next.previous. Finally, I decremented the list's size by 1 and returns the value of the node that was removed. After this, the node (element) is garbage collected and removed.
The worst-case performance of this method happens if the length of the deque isn't 0 and remove_element_at is called. However, this would still work in constant time as the first value is removed which is easily accessed through the header node. Therefore, it never has to change based on the size of list and always does the same thing - making it run in constant time.

**peek_front:**

If the length of the deque is 0 it returns None. Otherwise, it calls the LinkedList method get_element_at.
This method returns the value of the node at the index passed by the user. The first thing I do is check the index value passed by the user. If the index passed is greater than or equal to the size of the list or less than 0, an IndexError is raised. This is to avoid any incorrect indices. If the index passed is correct and in the first half of the list (less than or equal to the size divided by 2), I create a variable called current at the header. Just like in insert_element_at, I loop forward through the list so that current is equal to the node right before the element that the user would like to see the value of. Then, I return the value of the next node. If the index passed is correct and in the second half of the list (greater than the size divided by 2), I create a variable called current, but at the trailer. I then walk backwards through the list so that current is equal to the node right after the element that the user would like to see the value of. Then, I move to the node before current and return its value.
The worst-case performance of this method is constant. This is because we always access the first node in the list. Therefore, it doesn't matter what the size of the deque is and this method runs in constant time.

**push_back:**

This method calls the LinkedList method append_element. This method adds a node to the tail position of the list and increases the size of the list by one accordingly. Importantly, this method is the only way of adding a node the tail position. First, the method creates a new node with a value equivalent to the value passed to it by the user, we'll call this node "newest". Then, it looks at the node in the current tail position, takes its next variable and sets it to point at newest. Then, the method takes newest's previous variable and sets it equal to the node that is in the tail position. After that, it sets newest's next variable and points it at trailer, while setting trailer's previous variable to point at newest. It then increments the size of list by 1 as 1 new node as has been appended in the tail position.
The performance of this method is also O(1) or constant time. This is because, no matter what the input is, this method works in the same way every time because it just slides the value between trailer and the previous tail node.

**pop_back:**

If the length of the deque is 0 it returns None. Otherwise, it calls the LinkedList method remove_element_at for the index size minus 1 as that is the index of the value at the back of the deque.

This method removes and returns the value of the node at the index. The first thing I do is check the index value passed by the user. If the index passed is greater than or equal to the size of the list or less than 0, an IndexError is raised. This is to avoid any incorrect indices. If the index passed is in the second half of the list (indices greater than the size divided by 2), I create a variable called current at the trailer. It starts at the trailer position and walks backward to the node right after the index we need to remove the node. I then create a new variable called element which is the node I'd like to remove It is accessed by calling current.previous. I then set current.previous to element.previous and current.previous.next to current so that element is no longer pointed to by either node. Finally, I decrement the size by 1 and return the value of element as the spec requires.
Otherwise, if the index is in the first half of the list, (less than or equal to the size divided by 2), I again create a variable called current at the header. Just like in insert_element_at, I loop forward through the list so that current is equal to the node right before the element that the user would like to remove. Then I create a variable called element which is the node that the user would like to remove. Then, I set current.next equal to the node after the node that is removed. This accessed by using element.next. Then I set that node's previous value to point at current by using current.next.previous. Finally, I decremented the list's size by 1 and returns the value of the node that was removed. After this, the node (element) is garbage collected and removed.
The worst-case performance of this method is also constant time. This is because we are always removing the node next to the trailer which is easily accessed. Removing the last node is always the same amount of time and never changes depending on the size of the deque, therefore this method operates in constant time.

**peek_back:**

If the length of the deque is 0 it returns None. Otherwise, it calls the LinkedList method get_element_at for the index size minus 1 as that is the index of the value at the back of the deque.
This method returns the value of the node at the index passed by the user. The first thing I do is check the index value passed by the user. If the index passed is greater than or equal to the size of the list or less than 0, an IndexError is raised. This is to avoid any incorrect indices. If the index passed is correct and in the first half of the list (less than or equal to the size divided by 2), I create a variable called current at the header. Just like in insert_element_at, I loop forward through the list so that current is equal to the node right before the element that the user would like to see the value of. Then, I return the value of the next node. If the index passed is correct and in the second half of the list (greater than the size divided by 2), I create a variable called current, but at the trailer. I then walk backwards through the list so that current is equal to the node right after the element that the user would like to see the value of. Then, I move to the node before current and return its value.
The worst-case performance of this method is constant time. This is because we are always accessing the tail node so we always go through the trailer to the node behind it.

## Stack:

**__init__:**

This method works in constant time and it creates the variable self.__dq which creates a deque (calling from either Array Deque or LinkedList Deque depending on the variables passed in the get_deque parameters). This method works in constant time as it does one thing every time that works exactly the same the entire time.

**__str__:**

This method works in linear time for both the Array Deque and LinkedList Deque. If it calls LinkedList Deque's string method it works in linear time because it goes through every number in the deque to output a string. Similarly, Array Deque's string method works in linear time as it walks through every value in the deque to output a string representation.

**__len__:**

This method's worst-case performance is constant time as the length methods for both Array Deque and LinkedList Deque return their size variables and work in constant time.

**push:**

This method pushes a value to the top of the stack and calls the push_front method the deque. In the worst-case scenario this method works in linear time. This is the case if the user is using Array Deque and the __grow method is called because the array had to be expanded because it ran out of space.

**pop:**

This method always calls the deque's pop_front method. The worst case performance of this method is constant time. This is because for both the Array Deque and LinkedList Deque the pop_front methods have a worst case performance of constant time. In their sections above I expand upon why these methods work in constant time.

**peek:**

This method always calls the deque's peek_front method. This method always works in constant time as both Array Deque and LinkedList's Deque methods run in constant time as explained above.

## Queue:

### __init__:

This method works in constant time and it creates the variable self.__dq which creates a deque (calling from either Array Deque or LinkedList Deque depending on the variables passed in the get_deque parameters). This method works in constant time as it does one thing every time that works exactly the same the entire time.

### __str__:

This method works in linear time for both the Array Deque and LinkedList Deque. If it calls LinkedList Deque's string method it works in linear time because it goes through every number in the deque to output a string. Similarly, Array Deque's string method works in linear time as it walks through every value in the deque to output a string representation.

### __len__:

This method's worst-case performance is constant time as the length methods for both Array Deque and LinkedList Deque return their size variables and work in constant time.

### enqueue:

This method calls push_back to push a value into the queue. The worst case performance of this method is if the user is using Array Deque and calls __grow to expand the size of the array. That would make the performance of this array O(n) as discussed above.

### dequeue:

This method always calls the deque's pop_front method. The worst-case performance of this method is constant as both Array Deque's and LinkedList Deque's pop_front methods run in constant time in the worst-case scenario as discussed above.

## Testing:

### Deque:

First, I tested my deque implementations. I tried both implementations, testing both the Array and LinkedList Deques with all my test cases. Both implementations passed every test. My main strategy was to test each method thoroughly and make sure that all methods worked as expected, calling the string representation to make sure the deque was the same as expected.

An example of this testing is peeking both front and back sides of the deque, popping to front and back, and pushing to both front and back. Each time, I called the string representation of the deque to make sure the deque looked as expected. Additionally, I tested how methods interacted with the length method  - making sure that the only variables that changed the length of the deque were the ones that were supposed to change the length of the deque. An example of this was popping values off and make sure that the length of the deque decreased (ex. test_nothing_goes_wrong_deque4). I also wanted to test the __grow method was working properly and didn't show the unused parts of the deque in the string representation. To check this, my strategy was to push values into the deque and make sure that the string representation was correct (ex. test_deque_grow4).  I also checked that the value returned by the pops were the correct values.
I also tested important edge cases. An example is peeking the front of an empty list. This accomplishes making sure that the deque returns None when it is empty. I checked this for pops too. I also tried printing an empty deque to make sure it was correct.

**Queue:**

I then tested my queue implementation. The first thing I did was test edge cases of an empty queue, checking the string representation and length were correctly returned.
Then, I checked that the enqueue and dequeue methods were working correctly and that they correctly influenced the string representation and lengths of my queue.  I also checked that the value returned by dequeue is the correct value.
Finally, I tried another edge case where I dequeued from an empty queue. My queue correctly returned None.

**Stack:**

I then tested my stack. I first tested edge cases again, trying pop, printing the string representation, and finding the length of an empty stack. I then tested push and pop methods to make sure they correctly impacted the string representation and length of the stack. I also checked that the value returned by pop is the correct value.

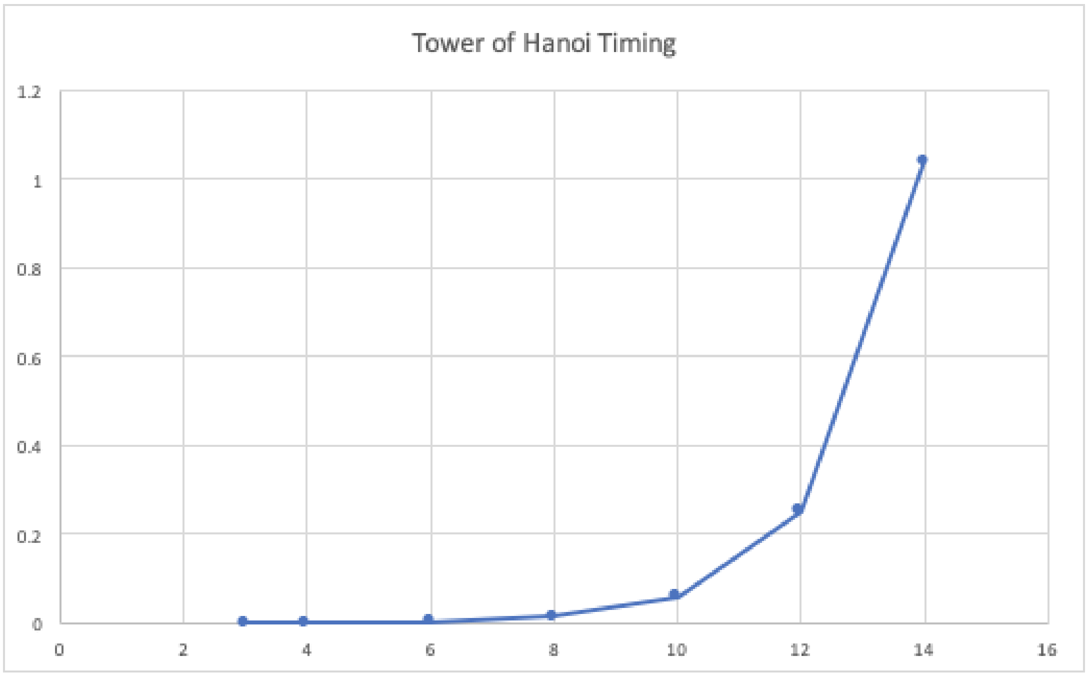
**On exceptions:**

I think it's not good to not raise exceptions as users can run into errors and not be sure why they happened without strong exception coverage. One example where I think raising exceptions could be useful is it users try to peek empty deques, stacks, or queues. I think this would help them quickly understand the deque, stack, or queue is empty rather than just returning None. This application is limited in the amount of methods accessible to the user and most of them don't really have great applications for raising exceptions (ex. push_back), but I think in general exceptions are a great way of getting a users attention to something important that is happening because of their actions and helping them understand what is happening in the program.

## Delimiter Check:

First, I open the file and loop through the lines, followed by looping through the characters. If a character is an opening bracket, parentheses, or curly brace, I push it onto the stack. If it's a closing bracket, parentheses, or curly brace, I pop the most recent value off of the stack and see if it's an opening bracket, parentheses, or curly brace. If it isn't then I return false. Also, the character is a closing bracket, parentheses, or curly brace and the length of stack is 0 so there is no opening character to match it, I return false as that means the file has imbalanced delimiters. Otherwise, one we break out of the loop, I check if the stack is empty. If it isn't, meaning that the file didn't have as many closing brackets, parentheses, or curly braces as opening brackets, parentheses, or curly braces, I return false. Otherwise, it returns true.
I tried this on a variety of files and it worked! I tried it also on files that have delimiters on separate lines to make sure that new lines wouldn't mess up my program. Fortunately, it worked!

## Tower of Hanoi:



In the graph above, I show the timing of my Tower of Hanoi file. The x-axis represents the number of discs, while the y-axis is the time it took for the program to run.
Similar to the Fibonacci example shown in class, the program rapidly becomes very slow as the number of inputs increases. Even though the pop and push function work most of the time in constant time the program still slow due to the large number of discs. This is because of the nature of recursion and the program calling itself twice in the else clause. Because of this for every single call except for the base case, the program runs two versions of itself which each call two additional versions of their own selves, continuing until the program reaches a base case. Because of this, the worst-case performance of this program is exponential.
The logic behind the Hanoi problem is (with 3 discs): the base case is that the we move the smallest disc (disc 0) by popping it from the source stack and pushing it to the destination stack. Otherwise, if the base case isn't reached (n > 0), the function calls itself decremented by one to signal its controlling the smaller disc where the destination stack is swapped with the auxiliary stack. In the example of three discs, this is done so that the function deposits the second to largest disc on the auxiliary stack. Once that call reaches the base case, the smallest disc is popped and pushed to the destination stack. This then calls the source stack to be popped and the value to be pushed to the destination stack, moving the 1 to the old destination (now new auxiliary stack location). Finally, it calls another instance of the function with n decremented by 1 once again, except this time the source stack is in the middle with the auxiliary stack to the left and the destination stack to the right. This then reaches the base case and pushes the 0 on top of the 1. This returns back to original Hanoi function call popping the 2 to the destination stack, 1 to the destination stack, and finally the 0 on top to the top of the destination stack.