Writeup for Project 3

Part 1: Explanation of Algorithm and Performance

This Program constructs two deque implementations: one using a linked_list to store its data and one using an array. Then the program further constructs a stack implementation and a queue implementation base on the deque. The following is a brief explanation of the algorithm and an analysis of the performance of each method.

(1) Algorithm and Performance of Linked_List_Deque

First, take a look at the Deque implementation using a linked_list. The Linked_list_deque implementation initially sets up an empty linked list for future storage of the data. The front of the deque is stored at the head position of the linked_list and the back of the deque is stored at the tail position of the linked list. So the string representation of the deque is the same as the string representation of the Linked_list which stores the data. For example, a deque with string representation [1, 2, 3] would be stored as a liniked_list with string representation [1. 2, 3]. So the string method of a deque simply returns the string representation of its corresponding Linked_list. As discussed in the writeup for Linked_list in project 2, the string method for Linked_list has linear time O(n) performance, so the string method for deque also has a linear time O(n) performance. Similarly, the len method of a deque returns the length of the Linked_list which stores the data. As the size of the linked_list is always stored, the len function has a constant time performance in both Linked_list and deque.

The push_front method of a Linked_list_deque adds one element to the front of the deque and increase the size by 1. To achieve this, simply add the element to the front of the Linked_list storing the data. When the deque has size 0, use the append_element method of the Linked_list as the insert_element_at method would raise an index error when the size is 0. If the size is larger than 0, use the insert_element_at method to insert the new element at index 0 in the Linked_list that stores the data. Then let's evaluate the performance of the push_front method. The append_element method always has constant-time performance. The insert_element_at method has a constant time performance when inserting at the head position as it simply connects the new node to the node at the head position and then the header node to the new node. So the push_front method which consists of these two methods also has a constant time performance.

The pop_front method removes the element at the front position in a deque and returns its value. When the size of the deque is 0, return and do nothing as no value is stored in the list. This is a special case as if we call the remove_element_at method to remove data from the storage media, an index error would be raised. When the size of the deque is larger than 0, call the remove_element_at method to remove the element at position 0 in the Linked_list that stores the data, return the value returned by the remove_element_at method. The pop_front method would also have constant time performance as although the remove_element_at method has linear time performance, the performance of removing an element at the head position is constant. The method simply links the header node to the second node in the Linked_list and the data originally at index 0 would be unlinked.

The peek_front method of a Linked_list_deque returns the value stored in the front position without changing the list. When the size of the deque is 0, return and do nothing as no value is stored in the list. This is a special case as if we use the get_element_at method to try to retrieve data from the storage media which is the Linked_list, the method would raise an index error. When the deque is non-empty, use the get_element_at method for Linked_list to retrieve the first data in the Linked_list, return its value. The peek_front method of a deque using the Linked_list as a data storage media would have constant time performance as the get_element_at method has constant time performance when retrieving data at index 0. Get_element_at method simply goes from the header node to the first element in the Linked-list and returns its value.

The push_back method adds a new element to the back of the deque and increases the size of the deque by 1. To achieve this, simply call the append_element method to append a new element to the Linked_list that stores the data. This method also automatically updates the size. The new element would be at the tail position of the Linked_list thus would be at the back of the deque. As the append_element function has a constant time performance, the push_back method also has a constant time performance.

The pop_back method removes the element at the back of a deque and returns its value. When the length of the deque is 0, return and do nothing. This is a special case as if we call the remove_element_at method to remove data from the storage media, an index error would be raised. When the size of the deque is non-zero, call the remove_element_at method to remove the element at the tail position in the Linked_list that stores the data, return the value returned by the remove_element_at method. The pop_back method would also have constant time performance as although the remove_element_at method has linear time performance, the performance of removing an element at the tail position is constant. As the tail position is in the second half of a Linked_list, the method would start from the trailer node to locate the target element. In this case, the method simply links the trailer node to the second last node in the Linked_list and the data originally at the tail position would be unlinked.

The peek_back method returns the value of the element at the back position in a deque, leaving the deque unchanged. When the size of the deque is 0, return and do nothing as no value is stored in the list. This is a special case as if we use the get_element_at method to try to retrieve data from the Linked_list that stores the data, the method would raise an index error. When the deque is non-empty, use the get_element_at method to retrieve the last data in the Linked_list, return its value. The peek_back method of a deque using the Linked_list as a data storage media would have constant time performance as the get_element_at method has constant time performance when retrieving data at the tail position. Get_element_at method simply goes from the trailer node to the last element in the Linked-list and returns its value.

(2) Algorithm and Performance of Array_Deque

The Array_deque implementation uses an array to store data for a deque. It has several attributes. Self.__capacity records the capacity of the array that stores the data. Self.__contents represents the array that stores the data. Self.__front and self.__back records the position of the front and back of the Deque stored in the array. Front grows to the left of the array and back grows to the right of the array. Self.__size stores the size of the deque. To save storage, the array is designed to be a circular array. When a piece of data is inserted, if it reaches one end of the array and it is full, it would wrap around and goes to the other end for unused spaces. If the whole array is full, a function would be called and the length of the array would be grown. For example, when element ‘name’ is inserted to back of the array [None, None, 3] where None represents that the cell is unoccupied, the new array would be [‘name’, None, 3].

The string method returns the string representation of the deque. When it is called, initialize three variables: string, current, and number_of_elements. String stores the string representation to be returned. Current stores the current location in the array and is initially set to be equal to self.__front. Number of elements stores the number of elements in the deque for other use. This function iterates n times through every element in a deque with size n, append the string representation to the variable string and its value after the iteration. In each iteration, the method first evaluates whether variable number_of_elements is greater than one. If it is greater than one, the method would also append a ‘,’ to separate the elements. Then the string method appends a blank space to the variable string for formatting. Finally, it appends the string representation of the element at index current. If variable current is larger than the capacity of the array storing the information, which means part of the data has wrapped around to the front of the array. Append the elements at index (current module capacity) to variable string. Finally, increase current and number_of_elements by 1 so that variable current goes to the index of the next element. After n iteration (where n is the size of the deque), add the close bracket ‘]’ to variable string return it for the string representation of the whole deque. This method has a linear time performance as it iterates exactly n time for a deque with size n in order to get all the elements in the string. For example, the array is [1, 2, 3] and self.__size is 3, front is at index 0 and back is at index 2, the method would first go to 1, append it to variable string(now string = ‘[1’) and do the same to element 2 and 3(now string = ‘[1, 2, 3’). Finally, it would add the ‘]’ so the string is ‘[1, 2, 3].

The len function simply returns the self.__size attribute stored in the memory, so it has constant-time performance.

The grow method is called to double the capacity of the array that stores the data. It first creates an empty array that is twice the capacity of the original one. Then it adds each element of the original array to the new array starting from index 0. This has the benefit of quickly locating the new front and back and make a timely update. The original data would occupy the first half of the new array. The new head would be at index 0 and the new back would be at index (self.__size - 1). Then the new array is ready for further operations. This method has linear time performance as it adds the elements in the old array to the new array one by one, which means it has to iterate through every element in the old array, resulting in linear time performance. For example, the current array is [1,2] with front of the deque at index 1 and tail at index 0. If grow method is called, the new array would be [None,None,None,None] and then [2,None,None,None] and finally [2,1,None,None] which takes 3 steps.

The push_front method adds one element to the front of the deque. If the size of the deque equals the capacity of the array that stores the data, call the grow function to generate an array containing the original data but with double capacity. Then add the new element to the array. As the front is at index 0. So the new element would be added to index (self.__capacity - 1) which is the last position in the array. Index (self.__capacity - 1) would be the new front and the size will be increased by 1. If the size of the deque is within the capacity of the array, add the element to the cell in front of the front index which is index ((front-1) module self.__capacity). Use modules arithmetic here so that if the front is at index 0, the method would automatically add the element to the last cell in the array. Then increase the size by 1. The new front would be at index ((front-1) module self.__capacity). This method has worst-case linear performance as the grow function has a linear performance. If the size of the deque equals the capacity of the array that contains it, grow function would be called which results in linear performance. In cases where the size is smaller than the capacity. The method has constant performance as it simply modifies one cell in the array.

The pop_front method pops the first element in the deque and returns its value. If the size of the deque is 0, the method would return and do nothing as no data is stored. If the size of the deque is non-zero. Store the value at the front index as to_return. Update the new front. As the array is circular, the new front would be ((front+1) module self.__capacity) so that the front of the deque would wrap around and go to the front position of the array if it reaches the end. By doing this, the poped element would be excluded from the data structure. Finally, decrease the size by 1 and return to_return so that the element poped will be returned. This method has constant time performance as it only returns a stored value which can be immediately accessed with a given index. Updating the front index and the size also only requires arithmetic thus is constant time. So the overall performance is constant time.

The peek_front method returns the value of the element at the front position without changing the deque. If the size of the deque is 0, the method would return and do nothing as no data is stored. If the size of the deque is non-zero. Store the value at the front index as to_return, return its value. This method has constant time performance as it only returns a stored value which can be immediately accessed with a given index.

The push_back method adds one element to the back of the deque. If the size of the deque equals the capacity of the array that stores the data, call the grow function to generate an array containing the original data but with double capacity. Then add the new element to the array. As the back is at index (self.__size -1). So the new element would be added to index (self.__size). Index (self.__size) would be the new back and the size will be increased by 1. If the size of the deque is within the capacity of the array, add the element to the cell after the back index which is index ((back+1) module self.__capacity). Use modules arithmetic here so that if the back is at index (self.__capacity -1), the method would automatically add the element to the first cell in the array. Then increase the size by 1. The new back would be at index ((back+1) module self.__capacity). This method has worst-case linear performance as the grow function has a linear performance. If the size of the deque equals the capacity of the array that contains it, grow function would be called which results in linear performance. In cases where the size is smaller than the capacity. The method has constant performance as it simply modifies one cell in the array.

The pop_back method pops the last element in the deque and returns its value. If the size of the deque is 0, the method would return and do nothing as no data is stored. If the size of the deque is non-zero. Store the value at the back index as to_return. Update the new back. As the array is circular, the new front would be ((back - 1) module self.__capacity) so that the back of the deque would wrap around and go to the back position of the array if it reaches exhausts all the cells in the front. By doing this, the poped element will be excluded from the data structure. Finally, decrease the size by 1 and return to_return so that the element poped will be returned. This method has constant time performance as it only returns a stored value which can be immediately accessed with a given index. Updating the back index and the size also only requires arithmetic thus is constant time. So the overall performance is constant time.

The peek_back method returns the value of the element at the back position without changing the deque. If the size of the deque is 0, the method would return and do nothing as no data is stored. If the size of the deque is non-zero. Store the value at the back index as to_return, return its value. This method has constant time performance as it only returns a stored value which can be immediately accessed with a given index.

(3) Algorithm and Performance of Stack

The Stack class is implemented with deque which is discussed above. It initially set up either a Linked_List_Deque or an Array_Deque specified by the user. Then all the Stack methods would be implemented using methods from deque. The top of the stack is stored to be the front of the deque. For example, a stack with string representation [1, 2, 3] has 1 as top and 3 as botton. Its corresponding deque representation would be [1, 2, 3].

The string method of stack returns the string representation of the deque that is storing the data. In both Linked_List_Deque and Array_Deque, getting the string representation has linear-time performance. So the performance of the string method for stack is also linear.

The len method of stack returns the size of the deque that is storing the data. In both Linked_List_Deque and Array_Deque, getting the size has constant-time performance. So the performance of the string method for stack is also constant.

The push method of stack pushes a new element to the top of the stack. To achieve so, add the new element to the front of the deque storing the stack because the top of the stack is at the front location. The push method has constant time performance if implemented by a Linked_List_Deque as the push_front method of Linked_List_Deque has constant-time performance. The push method would have linear time worst-case performance if implemented by an Array_Deque as the push_front method of Array_Deque has linear worst-case performance.

The pop method removes the element at the top of the stack and returns its value. Simply use the pop_front method to pop the first element in the deque that contains the data. This method has constant-time performance for both implementation as in both Linked_List_Deque and Array_Deque the pop_front method has constant-time performance.

The peek method returns the value of the element at the top of the stack, leaving the stack unchanged. As the top of the stack is positioned at the front of the deque that contains the data. Use the peek_front function to return the value of the front element. This method has constant-time performance for both implementation as in both Linked_List_Deque and Array_Deque the peek_front method has constant-time performance.

(4) Algorithm and Performance of Queue

The Queue class is implemented with deque which is discussed above. It initially set up either a Linked_List_Deque or an Array_Deque specified by the user. Then all the Queue methods would be implemented using methods from deque. The front of the queue is stored to be the front of the deque. For example, a Queue with string representation [1, 2, 3] has 1 as front and 3 as back. Its corresponding deque representation would be [1, 2, 3].

The string method of Queue returns the string representation of the deque that is storing the data. In both Linked_List_Deque and Array_Deque, getting the string representation has linear-time performance. So the performance of the string method for queue is also linear.

The len method of queue returns the size of the deque that is storing the data. In both Linked_List_Deque and Array_Deque, getting the size has constant-time performance. So the performance of the string method for queue is also constant.

The enqueue method of queue enqueues a new element to the back of the queue. To achieve so, add the new element to the back of the deque storing the data because the back of the queue is at the back location in the deque. The enqueue method has constant time performance if implemented by a Linked_List_Deque as the push_back method of Linked_List_Deque has constant-time performance. The enqueue method would have linear time worst-case performance if implemented by an Array_Deque as the push_back method of Array_Deque has linear worst-case performance.

The dequeue method removes the element at the front of the queue and returns its value. Simply use the pop_front method to pop the first element in the deque that contains the data. This method has constant-time performance for both implementation as in both Linked_List_Deque and Array_Deque the pop_front method has constant-time performance.

The peek method returns the value of the element at the front of the queue, leaving the queue unchanged. As the front of the queue is positioned at the front of the deque that contains the data. Use the peek_front function to return the value of the front element. This method has constant-time performance for both implementation as in both Linked_List_Deque and Array_Deque the peek_front method has constant-time performance.

Part 2: Explanation of Test Cases

As Deque is the basis for both Stack and Queue, its successful implementation is of great significance. The DSQ_Test File uses 150 test cases to test the correctness of implementation of every method in all the classes. In the setup function, create 3 objects deque, stack, queue for testing.

(1) Test for Deque

In the Test for the Deque class, each method is tested. The first part tests the string method for an empty deque. As no data is stored, the result should be an empty bracket ‘[ ]’. Then test the push_front method. Use the push_front method to push elements onto the deque, check whether the string representation after the push function matches the desired result. For example, if 1 is pushed front onto the deque and then 2 is pushed front onto the deque, then the string representation should be [2, 1]. Test separately with push_front function operating on deques with sizes 0, 1, 2 and 3 to make sure the method works on deque of all length.

Then Test the length method for deques. Use the push_front method to push elements onto the deque so that the length of the deque should be 1, 2, 3, 4 respectively. Test whether the returned length of the deque equals to these integers. Note that by testing the length method, it also shows that the push_front method works correctly by adding 1 to the size of the deque each time the function is called.

After this part, the push_back method is tested. Similarly, compare the string representation of the deque after the push function with the desired result. For example, if 1 is pushed_back onto the deque and then 2 is pushed_back onto the deque, the string representation should be [1, 2]. Also, check the length of the deque after the push_back function is called. The length should be increased by 1. Test separately on deque with length 0,1, 2, 3 to make sure the method works in every situation.

Test the peek_front method. When the size is 0, test that the method should return None as no data is stored in the deque. If the size is greater than 0, test that the value stored in the front position should be returned. Also, after the peek_front method is called, test with the length method and string method that the peek_front function does not change the deque and the length of the deque. For example, push_front 1 then 2 then 3 onto the deque, now the deque is [3, 2, 1]. The peek_front method should return value 3 and after the method is called, str(deque) should return [3, 2, 1] and len(deque) should return 3. Test separately for the peek_front method operating on deque with length 0, 1, 2 and 3 and 4.

Test the peek_back method. When the size is 0, test that the method should return None as no data is stored in the deque. The deque should remain to be [ ] with length 0. If the size is greater than 0, test that the value stored in the back position should be returned. Also, after the peek_back method is called, test with the length method and string method separately that the peek_back function does not change the deque and the length of the deque. For example, push_front 1 then 2 then 3 onto the deque, now the deque is [3, 2, 1]. The peek_back method should return value 1 and after the method is called, str(deque) should return [3, 2, 1] and len(deque) should return 3. Test separately for the peek_back method operating on deque with length 0, 1, 2 and 3 and 4.

Test the pop_front method. When the size is 0, test that the method should return None as no data is stored in the deque. The deque should remain to be [ ] with length 0. If the size is greater than 0, test that the element at the front position should be removed and its value returned. Use the length method and string method to test that the pop_front function removes the element at the front position and decreases the size by 1. For example, push_front 1 then 2 then 3 onto the deque, now the deque is [3, 2, 1]. The pop_front method should return value 3 and after the method is called, str(deque) should return [2, 1] and len(deque) should return 2. Test separately for the pop_front method operating on deque with length 0, 1, 2 and 3 and 4.

Test the pop_back method. When the size is 0, test that the method should return None as no data is stored in the deque. The deque should remain to be [ ] with length 0. If the size is greater than 0, test that the element at the back position should be removed and its value returned. Use the length method and string method to test that the pop_back function removes the element at the back position and decreases the size by 1. For example, push_front 1 then 2 then 3 onto the deque, now the deque is [3, 2, 1]. The pop_back method should return value 1 and after the method is called, str(deque) should return [3, 2] and len(deque) should return 2. Test separately for the pop_back method operating on deque with length 0, 1, 2 and 3 and 4.

(2) Test for Stack

In the Test for Stack, each method is tested to make sure they all function properly and interact with a Stack object correctly. First test that an empty stack should have string representation [ ]. Then test the push method acting on stacks. Use the push method to push elements onto a stack, check whether the string representation after the push method is called matches the desired result. For example, if 1 is pushed onto the stack and then 2 is pushed onto the stack, then the string representation should be [2, 1]. Test separately with push method operating on stacks with sizes 0, 1, 2 and 3 to make sure the method works on stacks of all lengths.

Then Test the length method for stacks. Use the push method to push elements onto the stack so that the length of the stack should be 0, 1, 2, 3 respectively. Test whether the returned length of the stack equals to these integers. Note that by testing the length method, it also shows that the push method works correctly by adding 1 to the size of the stack each time the function is called.

Test the peek method. When the size is 0, test that the method should return None as no data is stored in the stack. The stack should remain to be [ ] with length 0. If the size is greater than 0, test that the value at the top should be returned, leaving the stack unchanged. Use the length method and string method to test this. For example, push 1 then 2 then 3 onto the stack, now the stack is [3, 2, 1]. The peek method should return value 3 and after the method is called, str(stack) should return [3, 2, 1] and len(stack) should return 3. Test separately for the peek method operating on deque with length 0, 1, 2 and 3.

Test the pop method. When the size is 0, test that the method should return None as no data is stored in the stack. The stack should remain to be [ ] with length 0. If the size is greater than 0, test that the element at the top should be removed and its value returned. Use the length method and string method to test that the pop method removes the element at the top and decreases the size by 1. For example, push 1 then 2 then 3 onto the stack, now the stack is [3, 2, 1]. The pop method should return value 3 and after the method is called, str(stack) should return [2, 1] and len(stack) should return 2. Test separately for the pop method operating on deque with length 0, 1, 2, 3 and 4.

(3) Test for Queue

In the Test for queue, each method is tested to make sure they all function properly and interact with a queue object correctly.

First test that an empty queue should have string representation [ ]. Then test the enqueue method acting on queues. Use the enqueue method to enqueue elements onto a queue, check whether the string representation after the enqueue method is called matches the desired result. For example, if 1 is enqueued onto the queue and then 2 is enqueued onto the queue, then the string representation should be [1, 2]. Test separately with enqueue method operating on queues with sizes 0, 1, 2 and 3 to make sure the method works on queues of all sizes.

Then Test the length method for queues. Use the enqueue method to enqueue elements onto the queue so that the length of the queue should be 0, 1, 2, 3 respectively. Test whether the returned length of the queue equals to these integers. Note that by testing the length method, it also shows that the enqueue method works correctly by adding 1 to the size of the queue each time the function is called.

Test the peek method. When the size is 0, test that the method should return None as no data is stored in the queue. The queue should remain to be [ ] with length 0. If the size is greater than 0, test that the value at the front should be returned, leaving the queue unchanged. Use the length method and string method to test this. For example, enqueue 1 then 2 then 3 onto the queue, now the queue is [1, 2, 3]. The peek method should return value 1 and after the method is called, str(queue) should return [1, 2, 3] and len(queue) should return 3. Test separately for the peek method operating on deque with length 0, 1, 2 and 3.

Test the dequeue method. When the size is 0, test that the method should return None as no data is stored in the queue. The queue should remain to be [ ] with length 0. If the size is greater than 0, test that the element at the front should be removed and its value returned. Use the length method and string method to test that the dequeue method removes the element at the front and decreases the size by 1. For example, enqueue 1 then 2 then 3 onto the queue, now the queue is [1, 2, 3]. The dequeue method should return value 1 and after the method is called, str(queue) should return [2, 3] and len(queue) should return 2. Test separately for the dequeue method operating on deque with length 0, 1, 2, 3 and 4.

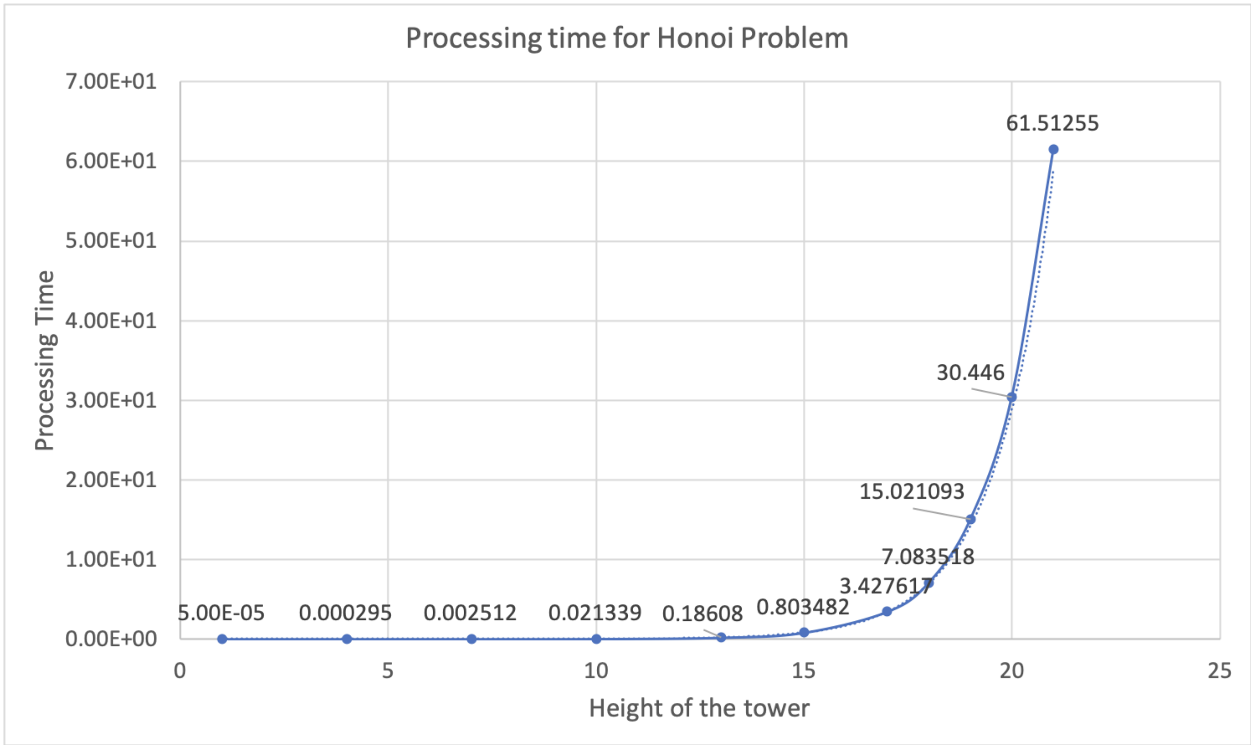
Note that to construct deque, stack, queues of different lengths, various permutations of different methods are called to ensure there are no hidden errors such as forgot to implement the front and back index. Some deques of length 3 are constructed by first pushing 5 elements into the deque, then poping 5 elements, then adding another 3 elements. All the methods in the Deque, Stack, Queue class are tested above to make sure each method returns the correct value and make correct modifications to the object they operate on. As the test would run separately in random order, passing those test cases suffice to show that the Classes and all its methods are working correctly.

Part 3: Discussion on whether to raise exceptions

Unlike the Linked_List project, all the methods in this project do not raise any exceptions. In the last project, all the methods raise an error when it receives a wrong index as input. These exceptions would stop the program from running. However, in this project, the user does not provide any indexes to the program. All they provide is the value stored in the structure. According to the design, the only methods that may raise exceptions are the peek and pop or dequeue methods as they are implemented using the remove_element_at and get_element_at functions in the Linked List. However here the problem can be resolved by simply letting the function to return. For example, if a user is popping from an empty stack, the function would just return and do nothing. This can serve to remind the user that the stack is empty which would be more appropriate than crashing the program.

From my personal perspective, the linked_list class serves as a media to store data for other structures thus should enable other users to track the potential issues resulting from methods while deque, stack, and queue are more like structures directly used by users thus should be more user-friendly.

Part 4: Performance of Hanoi



Height of Hanoi Tower	**1**	**4**	**7**	**10**	**13**	**15**	**17**	**18**	**19**	**20**	**21**
Processing Time (s)	5.00E-05	0.00029	0.00251	0.02133	0.18608	0.80348	3.42761	7.08351	15.0210	30.446	61.5125

The above graph is the processing time for Hanoi, the blue line represents the data observed and the dotted line is the exponential trend line for the data provided generated by Excel. It is clear that the performance of Hanoi is exponential. It can be further observed from the graph and table that the processing time almost double when the height of the tower increases by 1. So the performance of the Hanoi tower should be O(2^n). That is probably one of the drawbacks of recursion. Although it can provide convenience to code writers. The performance can sometimes be horrible.