Stacks and queues can be very useful data structures in the right scenarios. A stack is a data structure in which elements are stored in order and when removing an element from the stack we remove the element that was most recently added. A queue is similar to a stack except we remove the element that was first added. For both a stack and a queue there are many different ways to implement and I will be comparing three of them. First, an implementation from our textbook using a Python list. Second, an implementation using a linked list which consists of nodes that store the data for each element as well as a pointer to the next node.Finally, an implementation using the deque class imported from collections. I will be analysing and comparing the run times of each function for the different implementations of stacks and queues.

The following are pseudocode for my stack implementations.

Stack implementation from the textbook using a Python list:

```
class tStack:                                   O(1)
    def __init__(self):                         O(1)
        initialize self.items to an empty list  O(1)

    def isempty(self):                          O(1)
        if self.items is an empty list return true  O(1)

    def push(self, item):                       O(1)
        append item to self.items               O(1)

    def pop(self):                              O(1)
        pop the top item off of self.items      O(1)

    def peek(self):                             O(1)
        return the last item in self.items      O(1)

    def size(self):                             O(1)
        return the length of self.items         O(1)
```

Stack implementation using a linked list:

```
class Node:                                     O(1)
   def __init__(self,initdata):                 O(1)
      initialize self.data to initdata          O(1)
      initialize self.next to None              O(1)

   def getData(self):                           O(1)
      returns the data for self                 O(1)
```

```
    def getNext(self):                                  O(1)
        returns the next node                           O(1)

    def setData(self,newdata):                          O(1)
        set self.data to newdata                        O(1)

    def setNext(self,newnext):                          O(1)
        set self.next to be newnext                     O(1)

class Stack:                                            O(1)
    def __init__(self):                                 O(1)
        initialize self.top to None                     O(1)
        initialize self.size to 0                       O(1)

    def isempty(self):                                  O(1)
        if self.Size equals 0:                          O(1)
            return True                                 O(1)
        else:                                           O(1)
            return False                                O(1)

    def push(self,item):                                O(1)

        if self.top is None:                            O(1)
            set self.top to be Node(item)               O(1)
            add 1 to self.Size                          O(1)

        else:                                           O(1)
            initialize newnode to be Node(item)         O(1)
            set the next node after newnode to be self.top  O(1)
            set self.top to be newnode                  O(1)
            add 1 to self.Size                          O(1)

    def pop(self):                                      O(1)

        if self is empty:                               O(1)
            return None                                 O(1)

        else:                                           O(1)
            initialize temp to be self.top              O(1)
            set self.top to be the next node after self.top  O(1)
            set the next node after temp to None        O(1)
            subtract 1 from self.size                   O(1)
            return the data for temp                    O(1)
```

```
    def peek(self):                                          O(1)
        return the data for self.top                         O(1)

    def size(self):                                          O(1)
      return self.Size                                       O(1)
```

Stack implementation using deque:

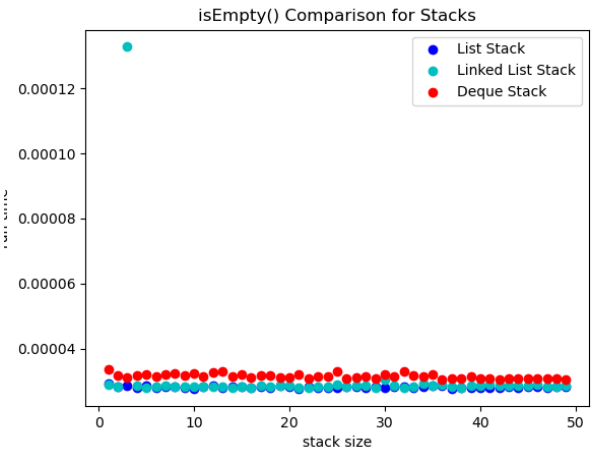
```
class Stack:                                                 O(1)
    def __init__(self):                                      O(1)
      initialize self.values to a deque()                    O(1)

    def isempty(self):                                       O(1)
      if self.values has elements                            O(1)
        return False                                         O(1)
      else:                                                  O(1)
        return True                                          O(1)

    def push(self, item):                                    O(1)
      self.values.append(item)                               O(1)

    def pop(self):                                           O(1)
      if self is empty:                                      O(1)
        return None                                          O(1)
      else:                                                  O(1)
        pop from self.values and return the popped value     O(1)

    def peek(self):                                          O(1)
        return the rightmost element in self.values          O(1)

    def size(self):                                          O(1)
      return len(self.values)                                O(1)
```

For each function a graph was created by timing the function call 100 times and taking the average for stack sizes of 1 to 50. The three different implementations were then plotted next to each other in order to compare run times.

isEmpty:

```
def isempty(self):                                  O(1)
        if self.items is an empty list return true  O(1)

def isempty(self):                                  O(1)
        if self.Size equals 0:                      O(1)
            return True                             O(1)
        else:                                       O(1)
            return False                            O(1)

def isempty(self):                                  O(1)
        if self.values has elements                 O(1)
            return False                            O(1)
        else:                                       O(1)
            return True                             O(1)
```

For the first implementation using a Python list, isempty is computed by checking whether the current list is equal to []. This operation happens in constant time. For the linked list implementation a size variable is kept and updated within the other functions. To check if the stack is empty we simply have to check if the size is equal to zero. Again, this happens in constant time. For the deque implementation we check whether or not there are elements in the deque. If not, return true. This also happens in constant time because a deque can be used as a boolean that returns true if it contains elements. For all three implementations isEmpty is O(1). As we can see in the graph to the right the isempty function for all three implementations happens in constant time and the difference in run time between them is quite small.

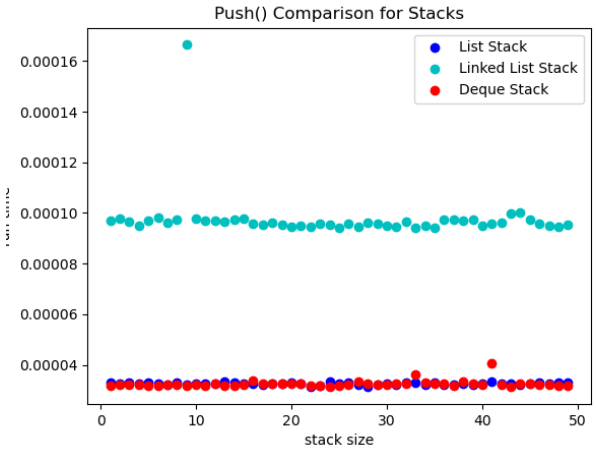


push:

```
def push(self, item):                               O(1)
        append item to self.items                   O(1)

def push(self,item):                                O(1)

        if self.top is None:                        O(1)
            set self.top to be Node(item)           O(1)
            add 1 to self.Size                      O(1)

        else:                                       O(1)
```

initialize newnode to be Node(item)	O(1)
set the next node after newnode to be self.top	O(1)
set self.top to be newnode	O(1)
add 1 to self.Size	O(1)
def push(self, item):	O(1)
self.values.append(item)	O(1)

To push an item in the list implementation we append that item to the list. Since Python lists are mutable appending an item happens in constant time so push is O(1). For the linked list implementation we have a variable self.top that points to the top of the stack. To push an item we create a new node of that item. Then, set that node to point to self.top as the next node. Finally, set self.top to be the new node and increase the size of the stack by 1. Through these operations we now have the pushed item on top of the stack pointing to the previous top as the next item. Due to our node implementation, all of these operations happen in constant time so push is O(1). For the deque implementation we use the append function to add the item on top of the stack. Deque appends items in constant time so push is O(1). The graph to the right clearly shows that all three functions are happening in constant time. The list and deque and list implementations are happening faster than the linked list because they both only have one operation in their push function whereas the linked list implementation has 4.

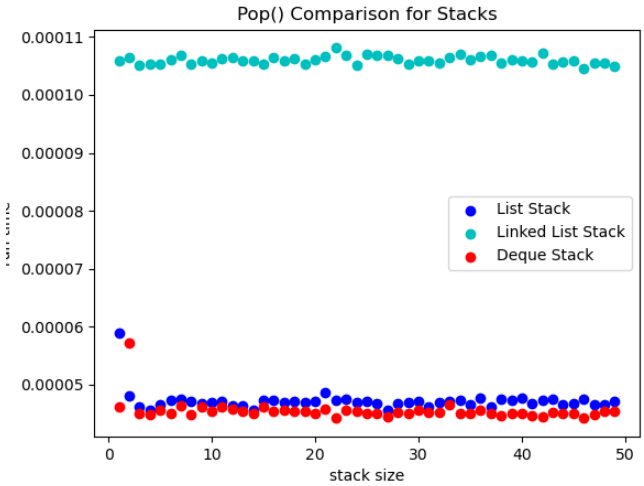


pop:

def pop(self):	O(1)
pop the top item off of self.items	O(1)
def pop(self):	O(1)
if self is empty:	O(1)
return None	O(1)
else:	O(1)
initialize temp to be self.top	O(1)
set self.top to be the next node after self.top	O(1)
set the next node after temp to None	O(1)
subtract 1 from self.size	O(1)

```
        return the data for temp                                O(1)

def pop(self):                                                  O(1)
    if self is empty:                                           O(1)
        return None                                             O(1)
    else:                                                       O(1)
        pop from self.values and return the popped value        O(1)
```

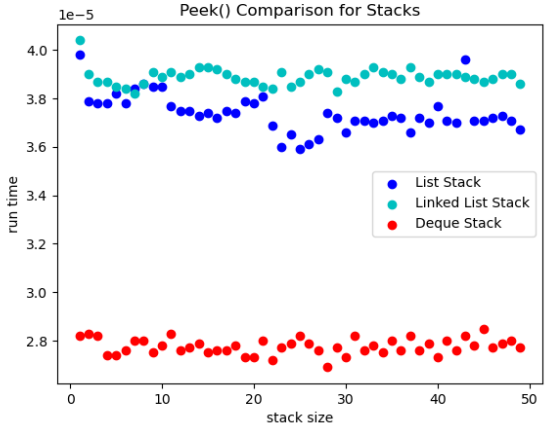
For the list implementation we use the list function pop to remove and return the item from the end of the list which is the top of the stack. Since we are working with the end of the list the function does not need to iterate through the list so pop is O(1). For the linked list implementation we make a temporary variable to store the top node. Then set the top node to be the next node. Then we set the temporary variable to point to nothing for the next node, decrease the size of the queue by one, and return the value stored in the temporary node. Similar to the push function, all of these operations happen in constant time so pop is O(1). Finally, for the deque implementation we use the deque function pop to remove and return the top value from the stack. The pop function in deque is O(1) so our pop function is O(1). The graph of these functions looks quite similar to the graph comparing the push functions. All three are O(1) with deque being the fastest and linked list being the slowest for the same reason as with push.



peek:

```
def peek(self):                                                 O(1)
    return the last item in self.items                          O(1)

def peek(self):                                                 O(1)
    return the data for self.top                                O(1)

def peek(self):                                                 O(1)
    return the rightmost element in self.values
    O(1)
```

For the peek function we want to return the top element from the stack without removing it. In the list implementation we can return the element in

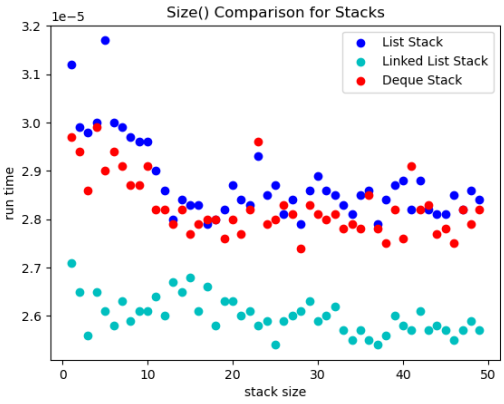


the last index of the list which is O(1) since python does not iterate through a list to find a certain index. For the linked list we can return the data corresponding to the top node which is also O(1). Finally, for the deque implementation we can return the element with the rightmost index in the deque which is O(1) similarly to indexing in a list. The graph matches this analysis, showing that all three functions are running in constant time. It appears that the deque implementation happens a bit faster than the list and linked list implementations.

size:

```
def size(self):                                  O(1)
    return the length of self.items              O(1)

def size(self):                                  O(1)
    return self.Size                             O(1)

def size(self):                                  O(1)
    return len(self.values)                      O(1)
```

For the list implementation we return the length of the list. For the deque implementation we return the length of the deque. For the linked list implementation we return the size variable that increments in the push and pop functions. The length function for list and deque are both O(1) and returning a value is O(1) so all three of these functions are O(1). The linked list size function happens slightly faster than the other two because all it has to do is return a variable that is already calculated. The list and deque implementations have to calculate the lengths. The graph of the function run times matches this analysis.



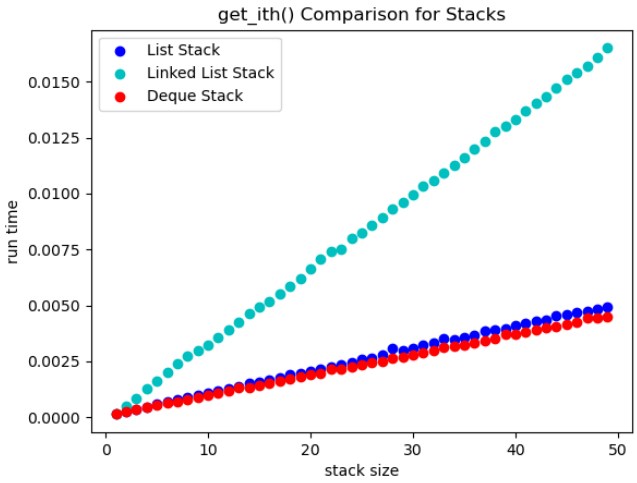
The main functions of a stack are to add and take away elements from the top of the stack. If we wanted to find an element that is in the middle we need a separate function. The get_ith function below takes a stack and returns the ith element by popping the first i-1 elements and adding them to a temporary stack. Then we use peek to return the ith element and add every element back from the temporary stack into the original.

```
def get_ith(stack, i):                                  O(1)
   if i is greater than the size of stack               O(1)
      return None                                       O(1)

   initialize temp to a Stack()                         O(1)
   for each element in stack up to the ith element:     O(i)
      pop from stack and push it onto temp              O(1)
```

```
    initialize toReturn to the top value in stack          O(1)
    for each element in temp:                              O(i)
        pop from temp and push that value onto stack       O(1)
    return toReturn                                        O(1)
```

Run time: O(1) + O(1) + O(1) + O(1) + O(n)(O(1)) + O(1) + O(n)(O(1)) + O(1)

The worst case run time happens when we are looking for the bottom element of the stack in which case i = n. The two loops repeat n times and they are not nested. Since push, pop, and peek are all O(1) for all three stack implementations, every other operation in the get_ith function is O(1). So the majority of the run time comes from these two loops and the run time becomes O(n). This graph shows the get_ith function for each implementation has a run time that increases as a linear function to the size of the stack. The linked list implementation's run time increases the fastest. This makes sense when looking back at the earlier graphs. Push and pop have longer run times in the linked list than the list or deque so as we increase the number of times we call each function the run time should increase faster.



Next, I will analyse functions of queues in the same way I analysed stacks. Below are pseudocode implementations of each queue.

Queue implementation from the textbook using a Python list:

```
class tQueue:                                              O(1)
    def __init__(self):                                    O(1)
        initialize self.items to an empty list             O(1)

    def isEmpty(self):                                     O(1)
        if self.items is an empty list return true otherwise false O(1)

    def enqueue(self, item):                               O(1)
        insert item to the back of the list                O(1)

    def dequeue(self):                                     O(1)
        pop from the front of the list and return the value O(n)

    def size(self):                                        O(1)
```

```
        return the length of self.items                          O(1)
```

Queue implementation using a linked list:

```
class Node:                                                      O(1)
    def __init__(self,initdata):                                 O(1)
        initialize self.data to initdata                         O(1)
        initialize self.next to None                             O(1)

    def getData(self):                                           O(1)
        returns the data for self                                O(1)

    def getNext(self):                                           O(1)
        returns the next node                                    O(1)

    def setData(self,newdata):                                   O(1)
        set self.data to newdata                                 O(1)

    def setNext(self,newnext):                                   O(1)
        set self.next to be newnext                              O(1)

class Queue:                                                     O(1)
    def __init__(self):                                          O(1)
        initialize self.front to None                            O(1)
        initialize self.back to None                             O(1)
        initialize self.Size to 0                                O(1)

    def isEmpty(self):                                           O(1)
        if self.Size equals 0:                                   O(1)
            return True                                          O(1)
        else:                                                    O(1)
            return False                                         O(1)

    def enqueue(self, item):                                     O(1)
        initialize temp to Node(item)                            O(1)

        if self.front is None:                                   O(1)
            set self.front to be temp                            O(1)
            set self.back to be temp                             O(1)
            add 1 to self.Size                                   O(1)

        else:                                                    O(1)
            set the next node after self.back to be temp         O(1)
            set self.back to be temp                             O(1)
            add 1 to self.Size                                   O(1)
```

```
  def dequeue(self):                                          O(1)

    if self is empty:                                         O(1)
      return None                                             O(1)
    else:                                                     O(1)
      initialize temp to be self.front                        O(1)
      set self.front to be the next node                      O(1)
      subtract 1 from self.Size                               O(1)
      return the data for temp                                O(1)

  def size(self):                                             O(1)
    return self.Size                                          O(1)
```

Queue implementation using deque:

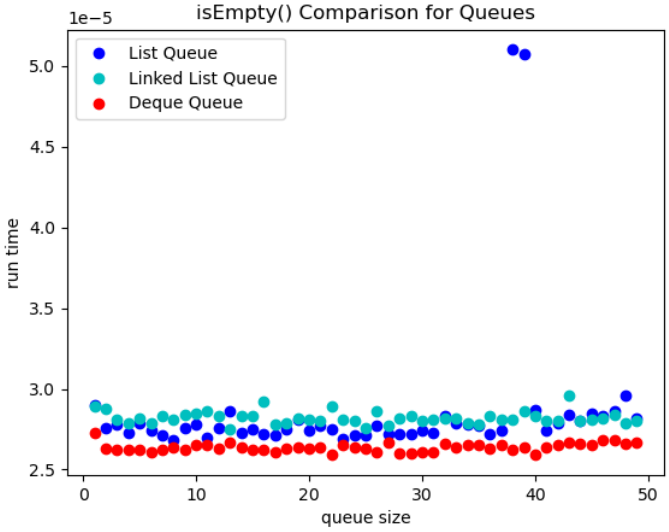
```
class dQueue:                                                 O(1)
  def __init__(self):                                         O(1)
    initialize self.values to a deque()                       O(1)
    initialize self.Size to 0                                 O(1)

  def isEmpty(self):                                          O(1)
    if self.values has elements:                              O(1)
      return False                                            O(1)
    else:                                                     O(1)
      return True                                             O(1)

  def enqueue(self, item):                                    O(1)
    append item to self.values                                O(1)

  def dequeue(self):                                          O(1)
    if self is empty:                                         O(1)
      return None                                             O(1)
    else:                                                     O(1)
      pop from the left of self.values and return the value   O(1)

  def size(self):                                             O(1)
    return the length of self.values                          O(1)
```

isEmpty:

```
 def isEmpty(self):                                           O(1)
    if self.items is an empty list return true otherwise false O(1)
```

```
def isEmpty(self):                                  O(1)
        if self.Size equals 0:                      O(1)
            return True                             O(1)
        else:                                       O(1)
            return False                            O(1)

 def isEmpty(self):                                 O(1)
        if self.values has elements:                O(1)
            return False                            O(1)
        else:                                       O(1)
            return True                             O(1)
```

The isEmpty functions for queues are implemented very similar to those for stacks. For the list implementation isempty is computed by checking whether the current list is equal to []. For the linked list implementation we store the size of the queue in a separate variable and increment it in the enqueue and dequeue functions. The isEmpty function simply checks if the size is equal to zero. For the deque implementation we check whether or not there are elements in the deque. If not, return true. Just like with the stack implementations isEmpty is O(1) for all three implementations. The graph of their run times show minor variations between the three functions. They are all running in constant time with the deque implementation running slightly faster than the other two.

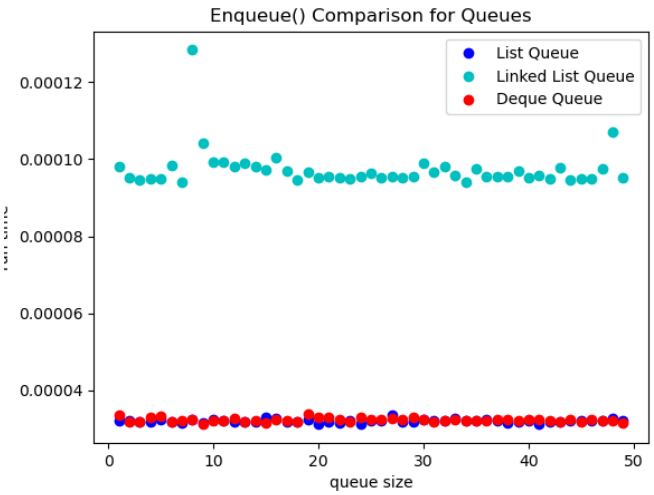


enqueue:

```
    def enqueue(self, item):                        O(1)
        insert item to the back of the list         O(1)

    def enqueue(self, item):                        O(1)
        initialize temp to Node(item)               O(1)

        if self.front is None:                      O(1)
            set self.front to be temp               O(1)
            set self.back to be temp                O(1)
            add 1 to self.Size                      O(1)
```

else:	O(1)
set the next node after self.back to be temp	O(1)
set self.back to be temp	O(1)
add 1 to self.Size	O(1)
def enqueue(self, item):	O(1)
append item to self.values	O(1)

The enqueue function is meant to add items to the queue. For both of my implementations I designed enqueue to add elements to the back and dequeue to remove elements from the front. Because of this, I slightly changed the textbooks queue class implementation so it would match the other two classes. So in order to enqueue an item I used the list append function to add the item to the back of the list which happens in constant time. So my first enqueue function is O(1). For the linked list implementation of a queue I needed to keep track of both the front and back nodes. Enqueue operates by creating a node of the new item, setting the current back node to point at the new node, and then making the new node the back of the queue and increasing the size by 1. All of these operations happen in constant time so enqueue is O(1) in the linked list implementation. Finally for the deque implementation we can use the append function to add the new item to the back of the deque. As mentioned earlier the deque append function is constant time so enqueue is O(1). Similar to the push and pop functions of stacks the graph clearly shows that the linked list implementation has a longer run time than the other two. Again. this happens because, even though they are all O(1), there are more operations happening in the linked list enqueue implementation.

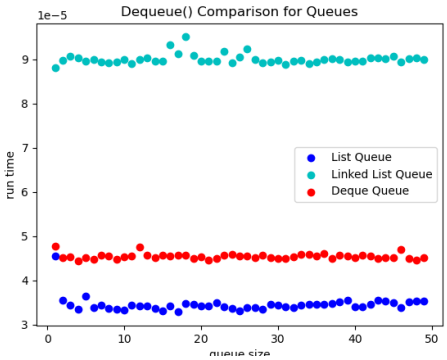


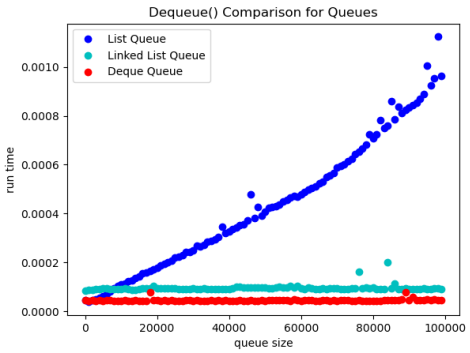
dequeue:

def enqueue(self, item):	O(1)
insert item to the back of the list	O(1)
def dequeue(self):	O(1)
if self is empty:	O(1)
return None	O(1)

else:	O(1)
initialize temp to be self.front	O(1)
set self.front to be the next node	O(1)
subtract 1 from self.Size	O(1)
return the data for temp	O(1)
def dequeue(self):	O(1)
if self is empty:	O(1)
return None	O(1)
else:	O(1)
pop from the left of self.values and return the value	O(1)

In each queue class the dequeue function is removing and returning the front element from the queue. In the first implementation I used the function pop(0) to remove the first element of the list. After removing the first element the rest of the list has to be shifted back 1 space which requires iterating through the whole list so pop(0) is O(n) which makes this version of dequeue O(n) as well. For the linked list class we create a temporary variable to store the front node. Then, we set the new front to be the next node in the queue. Lasty, we reduce the size by one and return the value from the temporary node. This removes the front node from the list and returns it's value. And since each function happens in constant time the dequeue function is O(1). For the deque implementation we want to remove elements from the left since enqueue is adding elements to the right. So we use the popleft function to remove and return the leftmost element in the deque. Deque is implemented so that adding and removing elements from either side is always O(1) which makes this dequeue function O(1) as well. For small queue sizes the first dequeue function does not increase in run time enough to be noticeable and it runs faster than the other two functions. As the size of the stack gets larger we can see more clearly that the deque and linked list implementations run in constant time and the list implementation's runtime increases as a linear function of the stack size.

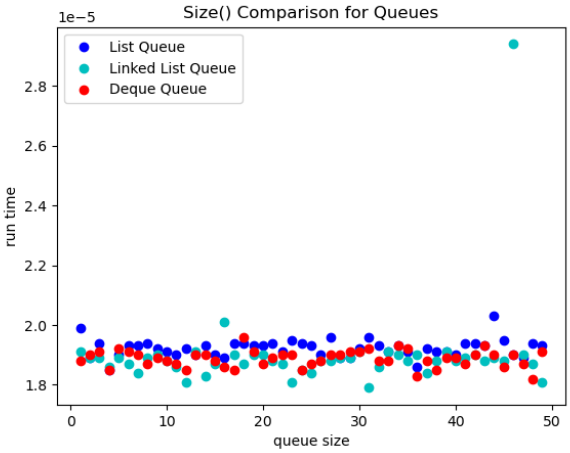




size:

def size(self):	O(1)
return the length of self.items	O(1)

```
def size(self):                                  O(1)
    return self.Size                             O(1)

  def size(self):                                O(1)
    return the length of self.values             O(1)
```



The size functions for the different queue implementations work the same way as the size functions for stacks. The list implementation can return the length of the list in constant time because Python lists store their size and don't have to iterate through the list to return the length. The linked list implementation can return the size in constant time because we are keeping a size variable that increments in the other functions. Lastly, the deque implementation can return the size by computing the length of the deque. The len function for deque is computed in constant time as well. So the size function for all three of the queue implementations is O(1). The graph here confirms that all functions are O(1) because the run times does not increase as the queue size increases.

In order to get the ith element of a queue we first dequeue every element up to the ith and enqueue them into a temporary queue. Then we can dequeue the ith element and save that as a new variable since that is the value we want to return. Next, enqueue the ith element into the temporary queue. The, we dequeue the remaining elements from the original queue and enqueue those elements into the temporary queue. This keeps the order correct so we can reverse the process and dequeue every element from the temporary queue and enqueue them to the original. Lastly, we return the ith element that we saved as a variable earlier in the function.

```
def get_ith(queue, i):                           O(1)
   if i is greater than the size of queue:       O(1)
        return None                              O(1)

   initialize temp to a Queue()                  O(1)
   for each of the first i-1 elements in queue:  O(i)
      dequeue from queue and enqueue it to temp  O(1) or O(n) for list implementation
```

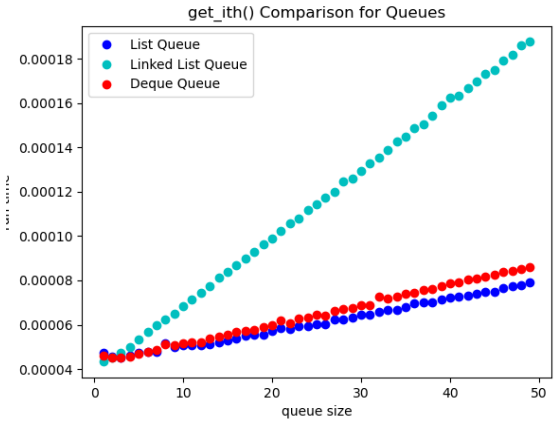
dequeue from queue and call that value toReturn	O(1)
enqueue toReturn to temp	O(1)
for the remaining elements in queue:	O(n-i)
dequeue from queue and enqueue it to temp	O(1) or O(n) for list implementation
for each element in temp:	O(n)
dequeue from temp and enqueue it to queue	O(1)
return toReturn	O(1)

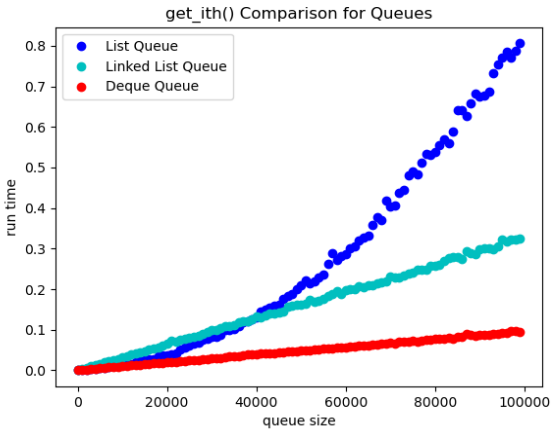
Run time:
O(1) + O(1) + O(1) + O(1) + O(n)(O(1)) + O(1) + O(1) + O(n)(O(1)) + O(n)(O(1)) + O(1) = O(n)

or

O(1) + O(1) + O(1) + O(1) + O(n)(O(n)) + O(1) + O(1) + O(n)(O(n)) + O(n)(O(1)) + O(1) = O(n^2)

The run time of get_ith changes depending on which queue implementation we are using. For the deque and linked list functions enqueue and dequeue are both O(1). But the list implementation has a dequeue function that is O(n). The majority of the run time comes from the three loops. For the dequeue and linked list get_ith functions the loops have constant run times and repeat at most n times so the run time is O(n). For the list get_ith function two of the loops have O(n) run time and repeat at most n times so the run time is O(n^2). At first it looks like all three functions have a linear run time. The list implementation is the fastest for small queue sizes. As the queues get larger we can see the quadratic growth rate of the get_ith function using the list implementation and the other two implementations are clearly linear.



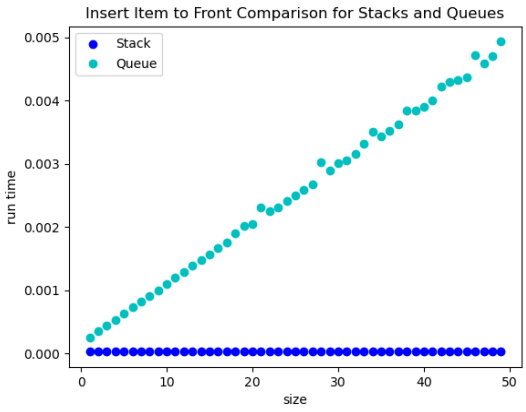


The last section of this report is focused on comparing the run times of certain operations between stacks and queues. For each operation I will create a graph of the run times as the size of the stack and queue get larger. I will be comparing the deque implementation for both stacks and queues because it has consistently lower run times than the other two options, and all of its operations are in constant time.

Inserting a new item to the top/front

```
def insertFront(queue, item):                          O(1)
    initialize temp to a Queue                         O(1)
    for each element in queue:                         O(n)
        dequeue from queue and enqueue it to temp      O(1)
    enqueue item to queue                              O(1)
    for each element in temp:                          O(n)
        dequeue from temp and enqueue it to queue      O(1)
```

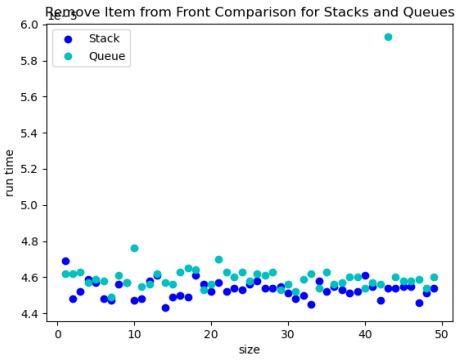
Run Time: O(1) + O(1) + O(n)(O(1)) + O(1) + O(n)(O(1))



To insert a new item to the top of a stack we simply use the push function which is O(1). Since my queue implementations add items to the back rather than the front a new function needs to be written to add items to the front. The above function takes a queue and an item and adds that item to the front of the queue. The way it works is similar to the get_ith function for queues. First, it dequeues every element from the queue. Then, enqueues the item to the queue. Lasty, it enqueues every item back to the queue. Both the first an the last step have to iterate through the entire queue so the loop repeats n times. This gives us a run time of O(n). This graph of the run times shows a constant time to remove an item from the top of a stack and a linear time to remove an item from the front of a queue.

Removing an item from the top/front.

Both stacks and queues have operations to remove an item from the top/front. Pop removes the top item off the stack in constant time and dequeue removes the front item from the queue in constant time. So this operation should be O(1) for stacks and



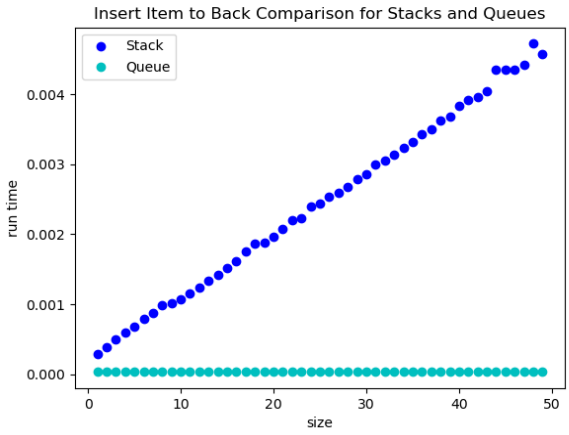
queues. That matches what we see in the graph to the right where the run time does not increase as the stack and queue sizes increase.

Inserting a new item to the bottom/back.

```
def pushBottom(stack, item):                          O(1)
    initialize temp to a Stack()                      O(1)
    for each element in stack:                        O(n)
        pop from stack and push that element onto temp  O(1)
    push item onto stack                              O(1)
    for each element in temp:                         O(n)
        pop from temp and push it onto stack          O(1)
```

Run time: O(1) + O(1) + O(n)(O(1)) + O(1) + O(n)(O(1)) = O(n)

When working with a queue we can add an item to the back using the enqueue function. The function pushBottom above is used to insert an item to the bottom of a stack. It works by popping every element off the stack, then pushing the item onto the empty stack, then pushing all of the original elements onto the stack. The majority of the run time comes from the two loops, both of which are O(1) repeated n times. So the pushBottom function is O(n). The graph demonstrates that as the sizes of the stacks and queues increase the runtime to insert an item to the back of a queue does not increase and the run time for a stack increases as a linear function of the stack size.



Removing an item from the bottom/back.

```
def popBottom(stack):                                 O(1)
    initialize temp to a Stack()                      O(1)
    for each element in stack except the last one:    O(n)
        pop from stack and push that element onto temp  O(1)
    pop the last element from stack                   O(1)
    for each element in temp:                         O(n)
        pop from temp and push that element onto stack  O(1)
```

Run time: O(1) + O(1) + O(n)(O(1)) + O(1) + O(n)(O(1)) = O(n)

```
def removeBack(queue):                                  O(1)
    initialize temp to a Queue                          O(1)
    for each element in queue except the last one:      O(n)
        dequeue from queue and enqueue it to temp       O(1)
    dequeue the remaining element from queue            O(1)
    for each element in temp:                           O(n)
        dequeue from temp and enqueue it to queue       O(1)
```

Run time: O(1) + O(1) + O(n)(O(1)) + O(1) + O(n)(O(1)) = O(n)

Neither stacks nor queues have operations that can remove an item from the bottom/back so a separate function was written for both of them. The first function, popBottom takes a stack and removes the element from the bottom. To accomplish this, every element except the last one is popped off the stack and pushed onto a temporary stack. Then, after the last element is removed, every element from the temporary stack is popped off and pushed onto the original stack. This leaves us with the original stack without the bottom element. The majority of the run is coming from the loops that iterate through the stack pushing and popping everything. Since both loops repeat n times popBottom is O(n). The function removeBack takes a queue and removes the back item. It works very similarly to popBottom but uses functions of a queue instead. First, it dequeues every element except the last one from the queue and enqueues those elements to a temporary queue. Then, it dequeues the remaining element from the queue. Finally, every item from the temporary queue is dequeued and enqueued back to the original queue. The removeBack function is O(n) for the same reason as popBottom. The graph of the run times shows the both functions increase in run time by a linear function of the stack/queue size confirming that both functions are O(n).