Assignment One – Nicholas Maisel

1 Queue.py Source Code

```
from LinkedList import Node
from LinkedList import LinkedList

class Queue(LinkedList):
    def __init__(self):
        self.head = None
        self.tail = None

    def isEmpty(self):
        if self.head == None and self.tail == None:
            empty = True
        else:
            empty = False
        return(empty)

    def enqueue(self,valueOfNode):
    #If the linkedlist is empty this makes the head and tail equal to the new node
        if self.isEmpty():
            n = Node(valueOfNode,None)
            self.head = n
            self.tail = n
        else:
        #if the linkedlist already has a node(s) you want to add to the tail
            n = Node(valueOfNode,None)
            self.tail.nextNode = n      # Adds to list
            self.tail = n

    def dequeue(self):
        if self.isEmpty():
            return(None)    #prevents an error if list is empty
        else:
            nodeToDequeue = self.head   #saves a copy of the node to dequeue
            if self.head.nextNode == None:  # checks if it is at end of list
                self.head,self.tail = None,None #resets list to empty if last node
            else:
                self.head = self.head.nextNode  #moves forward to next node
            return(nodeToDequeue)

    def peek(self):
        return(self.head.valueOfNode) #shows the value of the next node to be dequeued

```

1.1 Documentation of Queue.py

The Queue.py file begins by importing the Node and LinkedList classses from the LinkedList file. (Lines 2-3)

The Queue class inherits the LinkedList class as its base class (Line 6). Next we must define the two 'pointers' we use with our Queue class, we define head and tail (lines 8-9).

The Queue class has four different methods, isEmpty(), enqueue(), dequeue(), and peek().

–isEmpty() is defined on lines 11-16. The method works by checking to see if both head and tail are equal to None. This functionality will become more clear when we go over the dequeue() method.

–enqueue() is defined on lines 18-18. When this method is called you must pass a valueOfNode parameter. The method first checks to see if the queue is empty to see if the item to be en-queued needs to be attached to the tail or if it is going to be the first element. If it is the first element, the Node object is instantiated using the valueOfNode argument that was given. It then sets head and tail to the newly created node, thus marking the begining of the queue. However, if the Queue already has items then it performs the same action of creating a new Node and using the provided valueOfNode as its value. Except, instead of setting head and tail equal to the new node, it takes the current tail(say x), and assigns x.nextNode equal to the newly created Node(n), attaching it to the Queue. Finally the tail is then set to the newest node (n) and the method is complete.

–dequeue() is defined on lines 30-39. This method is used to, well, dequeue the first object in the Queue. It takes the first object because of the way queue's operate, using the FIFO mindset. The method starts on line 31 when it ensures that there is indeed items in the Queue to be de-queued, if not, the method simply returns None. If there are elements to be de-queued, the method continues. On line 34 the variable nodeToDequeue is introduced. nodeToDequeue is assigned to, and essentially holds a copy of the head node of the Queue. Next on line 35 we check to see if the nextNode value of the current head node is equal to None. This would denote the end of the list. If head.nextNode is equal to None, the list is empty after we dequeue the current nodeToDequeue, so we set the head and tail to None (ah, remember, if head and tail are equal to None, then the Queue is said to be empty)(line 36). If the nodeToDequeue is not the last Node in the Queue, then the method simply takes the current head and moves the head to the next node in the LinkedList. To move to the next node in the list we simply set head to the current head.nextNode. This also removes the nodeToDequeue from the Queue but, that is okay, we made a copy (nodeToDequeue)(line 38). Finally, the method ends by returning the nodeToDequeue, which will either be None or a Node. –peek() is defined on lined 41-42, and simply 'peeks' or reads the value of the current head Node, which is the next node to be dequeued.

2 Stack.py Source Code

```
from LinkedList import Node
from LinkedList import LinkedList

class Stack(LinkedList):        #this inherits the Node class
    def __init__(self):
        self.top = None


#The isEmpty function is used to make sure, before the linkedlist is edited
#that the edit will cause no errors.
    def isEmpty(self):
        if self.top:
            empty = False
        else:
            empty = True
            return(empty)

    def push(self,valueOfNode):
        if self.isEmpty():
            #Creates a new node to be pushed and attaches it to list with next node
            n = Node(valueOfNode,self.top)
            self.top = n               #Makes the new node the top of list for LIFO
        else:
            n = Node(valueOfNode,None)
            n.nextNode = self.top
            self.top = n


    def pop(self):
        if not self.isEmpty():         # Ensures an empty list isnt popped
            nodeToBePopped = self.top  # Makes a copy of the node to return
            self.top = self.top.nextNode   # Remove node from list by moving up
            return(nodeToBePopped)
```

2.1 Documentation of Stack.py

The Stack.y file begins by importing the LinkedList and Node classes from LinkedList.py (Lines 1-2).

The Stack class inherits the LinkedList class as its base class (Line 4). Next we must define the pointer we use with our Stack class, we define top on line 6, the top variable will be used to keep track of where the latest Node is.

The Stack class has three different methods isEmpty(), push(), and pop().

– isEmpty() is defined on lines 11-16. isEmpty() is used to ensure that the Stack is not empty. Since the top variable is used to keep track of the latest node added, if there are any nodes in the Stack then top will be defined. This method works by checking to see if top is defined, if not, then we say that the Stack is empty, else, the Stack has elements.

– push() is defined on lines 18-26. The method takes on argument, valueOfNode. The methods checks to see if the Stack is empty, if it is empty,it creates a new node with the value passed as valueOfNode. Another interesting note about line 21, is that it sets the nextNode value to self.top, this is because we want to mark it as the top of the Stack. Then on line 22, it sets top equal to the newly created Node. However, if the Stack is not empty, we move to line 24. We start by again, creating another new Node with the value that was passed as valueOfNode,however, this time, we set the nextNode value to None, since it is at the top and there are no more nodes.

– pop() is defined on lines 29-33. pop() starts by ensuring there are Nodes to be popped off of the stack, it does this on line 30. On line 31, we create a copy of the top Node and call it nodeToBePopped. To 'pop' the Node from the stack we take the current top's nextNode value, and assign it as the new top. Finally on line 33, pop() returns the nodeToBePopped Node.